

AD-A239 715 ITATION PAGE



Form Approved
OPM No. 0704-0188

four per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
s burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: 10 May 1991 to 01 Jun 1993	
4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report:UNISYS Corporation, UCS Ada, Version 1R1, 2200/600 (Host & Target), 910510S1.11161				5. FUNDING NUMBERS	
6. AUTHOR(S) National Institute of Standards and Technology Gaithersburg, MD USA					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Institute of Standards and Technology National Computer Systems Laboratory Bldg. 255, Rm A266 Gaithersburg, MD 20899 USA				8. PERFORMING ORGANIZATION REPORT NUMBER NIST90UNI515_1_1.11	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, RM 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) UNISYS Corporation, UCS Ada, Verison 1R1, Gaithersburg, MD, 2200/600 running OS1100, Verison 43R2 (Host & Target), ACVC 1.11.					
<div style="text-align: center;"> </div> <div style="text-align: right;"> 91-08767 </div>					
14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.				15. NUMBER OF PAGES	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT		

AVF Control Number: NIST90UNI515_1_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 10 May 1991.

Compiler Name and Version: UCS Ada, Version 1R1

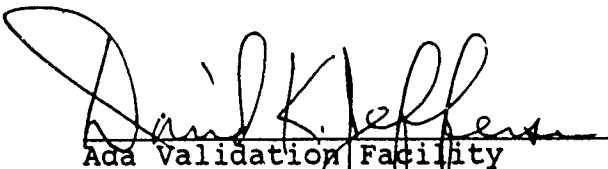
Host Computer System: 2200/600 running OS1100, Version 43R2

Target Computer System: 2200/600 running OS1100, Version 43R2

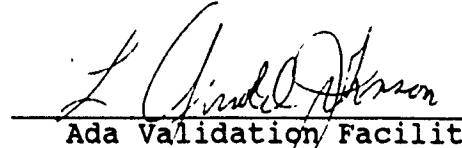
A more detailed description of this Ada implementation is found in section 3.1 of this report.

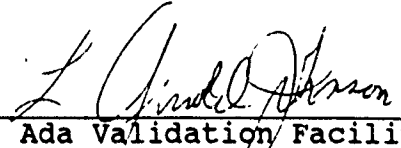
As a result of this validation effort, Validation Certificate 910510S1.11161 is awarded to UNISYS Corporation. This certificate expires on 01 March 1993.


This report has been reviewed and is approved.


Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)

Computer Systems Laboratory (CLS)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899


Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311


Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group


Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

AVF Control Number: NIST90UNI515_1_1.11
DATE COMPLETED

BEFORE ON-SITE: April 09, 1991

AFTER ON-SITE: May 10, 1991

REVISIONS: July 24, 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910510S1.11161
UNISYS Corporation
UCS Ada, Version 1R1
2200/600 => 2200/600

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

AVF Control Number: NIST90UNI515_1_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 10 May 1991.

Compiler Name and Version: UCS Ada, Version 1R1

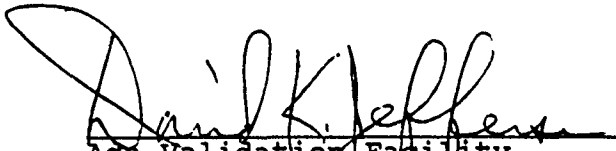
Host Computer System: 2200/600 running OS1100, Version 43R2

Target Computer System: 2200/600 running OS1100, Version 43R2


A more detailed description of this Ada implementation is found in section 3.1 of this report.


As a result of this validation effort, Validation Certificate 910510S1.11161 is awarded to UNISYS Corporation. This certificate expires on 01 March 1993.

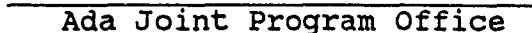
This report has been reviewed and is approved.


Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)

Computer Systems Laboratory (CLS)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899


Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group


Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311


Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: UNISYS Corporation

Certificate Awardee: UNISYS Corporation

Ada Validation Facility: National Institute of Standards and
Technology
Computer Systems Laboratory (CSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: UCS Ada, Version 1R1

Host Computer System: 2200/600 running OS1100, Version
43R2

Target Computer System: 2200/600 running OS1100, Version
43R2

Declaration:

I the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

J. S. Brown
Customer Signature

May 9, 1991
Date

J. S. Brown

Company: UNISYS Corporation

Title: Language Products Department Manager

J. S. Brown

May 9, 1991

Certificate Awardee Signature

Date

J. S. Brown

Company: UNISYS Corporation

Title: Language Products Department Manager

TABLE OF CONTENTS

CHAPTER 1	1-1
INTRODUCTION	1-1
1.1 USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2 REFERENCES	1-1
1.3 ACVC TEST CLASSES	1-2
1.4 DEFINITION OF TERMS	1-3
CHAPTER 2	2-1
IMPLEMENTATION DEPENDENCIES	2-1
2.1 WITHDRAWN TESTS	2-1
2.2 INAPPLICABLE TESTS	2-1
2.3 TEST MODIFICATIONS	2-4
CHAPTER 3	3-1
PROCESSING INFORMATION	3-1
3.1 TESTING ENVIRONMENT	3-1
3.2 SUMMARY OF TEST RESULTS	3-1
3.3 TEST EXECUTION	3-2
APPENDIX A	A-1
MACRO PARAMETERS	A-1
APPENDIX B	B-1
COMPILATION SYSTEM OPTIONS	B-1
LINKER OPTIONS	B-2
APPENDIX C	C-1
APPENDIX F OF THE Ada STANDARD	C-1

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued. Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3. For each Ada implementation, a customized test suite is produced

by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 93 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-03-14.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	C35702B	B41308B	C43004A	C45114A
C45346A	C45612A	C45612B	C45612C	C45651A	C46022A
B49008A	A74006A	C74308A	B83022B	B83022H	B83025B
B83025D	B83026B	C83026A	C83041A	B85001L	C86001F
C94021A	C97116A	C98003B	BA2011A	CB7001A	CB7001B
CB7004A	CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B
BD1B06A	AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A
CD2A41A	CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A
CD4022A	CD4022D	CD4024B	CD4024C	CD4024D	CD4031A
CD4051D	CD5111A	CD7004C	ED7005D	CD7005E	AD7006A
CD7006E	AD7201A	AD7201E	CD7204B	AD7206A	BD8002A
BD8004C	CD9005A	CD9005B	CDA201E	CE2107I	CE2117A
CE2117B	CE2119B	CE2205B	CE2405A	CE3111C	CE3116A
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 173 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113N..Y (12 tests)	C35705N..Y (12 tests)
C35706N..Y (12 tests)	C35707N..Y (12 tests)
C35708N..Y (12 tests)	C35802N..Z (13 tests)

C45241N..Y (12 tests)	C45321N..Y (12 tests)
C45421N..Y (12 tests)	C45521N..Z (13 tests)
C45524N..Z (13 tests)	C45621N..Z (13 tests)
C45641N..Y (12 tests)	C46012N..Z (13 tests)

C24113I..M (5 TESTS) contain lines that exceed this implementation's maximum input-line length of 132 characters.

The following 21 tests check for the predefined type `SHORT_INTEGER`; for this implementation, there is no such type:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, there is no such type.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for small that are not powers of two or ten; this implementation does not support such values for small.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

CD2A53A uses a value other than a power of two for 'SMALL. (See section 2.3.)

The 21 tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is

attempted.

CE2107B..E	CE2107G..H	CE2107L	CD2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 18 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B23004A	B24007A	B24009A	B28003A	B32202A	B32202B
B32202C	B37004A	B45102A	B61012A	B91004A	B95069A
B95069B	B97103E	BA1101B4	BC2001D	BC3009C	

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

J. S. Brown
P.O.Box 64942
St. Paul Mn. 55164
Phone: (612) 635-2077

For a point of contact for sales information about this Ada implementation system, see:

M. C. Adelman
8008 Westpark Dr.
McLean Va. 22102
Phone: (703) 556-5029

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3779
b) Total Number of Withdrawn Tests	93
c) Processed Inapplicable Tests	298
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point	

Precision Tests

0

- f) Total Number of Inapplicable Tests 298 (c+d+e)
- g) Total Number of Tests for ACVC 1.11 4170 (a+b+f)

When this implementation was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 298 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host / target computer.

After the test files were loaded onto the host / target computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

The options invoked by default for validation testing during this test were:

A. compiler default options invoked were:

NO-ALLOC, CACHE/D0, NO-CLEAR, CODE, SINGLESPEACE,
DYNAMIC/32768, EXTENDED, NO-GEN-INST, NO-LEVEL,
LIBUPDATE, NO-LINKINFO, MAXERRORS/100, NO-OBJECT,
NO-OBJ-PKT, NO-OPTIM, OPTIONS, NO-REMARK,
RUNCHECK, NO-SOURCE, STACK/64, NO-TRANSFORM, WARNING,
WIDE, NO-XREF-SAVE

B. specific options invoked for all tests were:

ADA-BUFFERS/1000, NO-OPTIONS, NO-DEBUG, TASKING, EXPAND,
NO-GEN-INL

C. other options invoked were as follows:

SOURCE - for tests requiring source list output (ie: B-tests, E-tests, and N/A tests that terminated in compile).

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is V the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	132
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
\$ACC_SIZE	36
\$ALIGNMENT	1
\$COUNT_LAST	262_143
\$DEFAULT_MEM_SIZE	16777216
\$DEFAULT_STOR_UNIT	36
\$DEFAULT_SYS_NAME	UCS
\$DELTA_DOC	2#1.0#e-35
\$ENTRY_ADDRESS	0
\$ENTRY_ADDRESS1	1
\$ENTRY_ADDRESS2	2
\$FIELD_LAST	132
\$FILE_TERMINATOR	
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"MAX_REC_NUM=>8"
\$GREATER_THAN_DURATION	100_000.0
\$GREATER_THAN_DURATION_BASE_LAST	131_073.0
\$GREATER_THAN_FLOAT_BASE_LAST	1.70E+38
\$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E308
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
\$HIGH_PRIORITY	1

\$ILLEGAL_EXTERNAL_FILE_NAME1	ADA*ABCDEFGHIJKLM.
\$ILLEGAL_EXTERNAL_FILE_NAME2	ADA*BDB*CDC.
\$INAPPROPRIATE_LINE_LENGTH	-1
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
\$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006E1.TST")
\$INTEGER_FIRST	-34359738367
\$INTEGER_LAST	34359738367
\$INTEGER_LAST_PLUS_1	34359738368
\$INTERFACE_LANGUAGE	UFTN
\$LESS_THAN_DURATION	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST	-131_073.0
\$LINE_TERMINATOR	ASCII.LF
\$LOW_PRIORITY	1
\$MACHINE_CODE_STATEMENT	NULL;
\$MACHINE_CODE_TYPE	NO_SUCH_TYPE
\$MANTISSA_DOC	35
\$MAX_DIGITS	17
\$MAX_INT	34359738367
\$MAX_INT_PLUS_1	34359738368
\$MIN_INT	-34359738367
\$NAME	NO_SUCH_TYPE_AVAILABLE
\$NAME_LIST	UCS
\$NAME_SPECIFICATION1	RITE*X2120A
\$NAME_SPECIFICATION2	RITE*X2120B
\$NAME_SPECIFICATION3	RITE*X3119A

\$NEG_BASED_INT	8#700000000001#
\$NEW_MEM_SIZE	0
\$NEW_STOR_UNIT	36
\$NEW_SYS_NAME	UCS
\$PAGE_TERMINATOR	ASCII.FF
\$RECORD_DEFINITION	NEW INTEGER;
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	36
\$TASK_STORAGE_SIZE	178
\$TICK	0.0002
\$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
\$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
\$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2
\$YOUR_PRAGMA	INTERFACE_NAME

APPENDIX B

COMPILATION SYSTEM OPTIONS

B.1 COMPILER OPTIONS:

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not this report.

Compiler Call Letter Options:

- D Prints the allocation listing; a listing of all program symbols for which storage is allocated.
- E Prints warning and remark messages.
- I Specifies that source input is from the runstream.
- L Invokes all of the listing options (same as keyword options ALLOC, LINKINFO, OBJECT, OBJ-PKT, OPTIONS, REMARK, SOURCE).
- N Specifies that no listing options are to take affect (opposite of the L option).
- O prints the object code listing.
- S Prints a listing of the source code.

Compiler Keyword Options:

ADA-BUFFERS/n

Controls the amount of storage used by the virtual memory of the compiler. You should only use this parameter when advised to do so by a Unisys support representative.

ALLOC/NO-ALLOC

Produces a listing of all symbols for which storage was allocated.

CACHE/spec

Used to improve wall time compilation performance by using the available real memory on the system to reduce the number of I/O requests.

CLEAR/NO-CLEAR

Controls whether or not uninitialized static variables are set

to zero.

CODE/NO-CODE

Controls production of object code.

DEBUG/spec

Controls the generation of PADS (Programmer's Advanced Debugging System) interface code.

DOUBLESPEACE/SINGLESPEACE

Defines the line spacing for the source program listing.

DYNAMIC/n

Controls the size of objects allocated in the dynamic stack (range is 0 to 32,768).

EXPAND/NO-EXPAND

Controls whether or not to insert code for subprograms that include the pragma **INLINE** and are not directly recursive.

EXTENDED/n

Defines the machine class on which the object module produced by the compiler executes n = blank - defaults to **LSS** definition:

- 0 - any extended mode machine
- 2 - 1100/90 and 2200/600 system
- 4 - 2200/200, 2200/300, and 2200/400 system

GEN-INL/NO-GEN-INL

Controls whether the compiler will place the code of a generic instantiation within the compilation unit (inline) or in a separate subunit.

GEN-INST/NO-GEN-INST

Controls whether or not the compiler will instantiate a pending uncompiled generic body whose template has been compiled or recompiled in another library.

LEVEL/STANDARD - NO-LEVEL

Controls whether or not to issue warning messages for nonstandard pragmas and attributes.

LIBUPDATE/NO-LIBUPDATE

Controls whether or not the compiler will produce an object module and update the program library.

LINKINFO/NO-LINKINFO

Controls the listing of external definitions, external references and logical bank sizes.

MAXERRORS/n

Specifies that the compiler terminate immediately if more than n major errors occur.

NARROW/WIDE

Controls the width of the printed listings (79 and 132 character lines respectively).

OBJECT/OBJECT-ONLY/NO-OBJECT

OBJECT	- produces an object listing with interspersed source lines.
OBJECT-ONLY	- produces an object listing with no interspersed source lines.
NO-OBJECT	- suppresses the object listing.

OBJ-PKT/NO-OBJ-PKT

Controls whether object code is additionally listed for I/O packets, parameter packets, and array descriptors.

OPTIM/NO-OPTIM

Controls whether or to increase the optimization performed by LSS.

OPTIONS/NO-OPTIONS

Controls whether or not options settings are listed.

REMARK/NO-REMARK

Controls whether or not REMARK class messages are listed during compile.

RUNCHECK/NO-RUNCHECK

Controls whether or not to generate extra code to perform range checks on array element references and substring expressions and terminate execution if error encountered.

SOURCE/NO-SOURCE

Controls whether or not a source listing is produced.

STACK/n

Specifies the maximum size of fixed, non-static objects that the compiler allocates in the UCS Ada fixed stack.

TASKING/NO-TASKING

Tells the compiler whether or not a library unit and its subunits use tasking declarations.

TRANSFORM/NO-TRANSFORM

Controls whether or not the compiler performs some transformations that cause improvements at execution-time but also can cause precision differences.

WARNING/NO-WARNING

Controls whether or not to list warning class messages of subclass USAGE or ARCHAIC-USAGE.

XREF-SAVE/NO-XREF-SAVE

Controls whether or not the compiler will store an intermediate version of the unit in the program library for use with the cross reference tool which produces a cross reference listing.

B.2 BINDER/LINKER OPTIONS:

The Binder/Linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to Binder/Linker documentation and not this report.

Binder Call Letter Options:

- B Perform a bind without automatically calling the LINK Processor.
- C Checks for binding errors without generating BINDER\$ and ADALINK\$ elements.
- I Causes the binder to read ADABIND commands from the runstream.
- E Causes the binder to print all error and warning messages issued by the binder and the linker.
- L Produces a long listing of the binding phase including:
 - list of the libraries used by the program
 - list of the units used by the program
 - list of the elaboration order of the unitsAlso causes the L option to be used by the LINK Processor.
- N Prevents nonfatal warning messages from being issued by the binder.
- S Causes the binder and the linking system to produce short listings.

There are no Keyword options for the binder or the Linker.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

type INTEGER is range -34_359_738_367 .. 34_359_738_367;

type FLOAT is digits 7

range -2#0.111_111_111_111_111_111_111_111#E+127 ..
2#0.111_111_111_111_111_111_111_111#E+127;

type LONG_FLOAT is digits 17

range -2#0.111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111#E+1023 ..
2#0.111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111_111111#E+1023;

type DURATION is delta 2#0.000_000_000_000_000_001# range
-86_400.0 .. 86_400.0;

end STANDARD;

Appendix F

Implementation-Dependent Language Features

The Ada language allows for certain machine dependencies that are described in the *UCS Ada Programming Reference Manual Volume 1*. Although no machine-dependent syntax or semantic extensions or restrictions are allowed, the standard does allow for the following implementation dependencies:

- Implementation-dependent pragmas and attributes
- The machine-dependent conventions that the *UCS Ada Programming Reference Manual Volume 1* describes in Section 13
- The restrictions on representation clauses that Ada allows

The purpose of this section, as required by the Ada language, is to describe each implementation-dependent characteristic of UCS Ada. As such, its subsections include the following:

- The form, allowed places, and effect of every implementation-generated pragma
- The name and the type of every implementation-dependent attribute
- The specification of the package SYSTEM
- The description of the representation clauses
- The conventions used for any implementation-generated name denoting implementation-dependent components
- The interpretation of expressions that appear in address clauses, including those for interrupts
- Any restrictions on unchecked conversions
- Any implementation-dependent characteristics of the input-output packages
- Characteristics of numeric types
- Other implementation-dependent characteristics

F.1. Pragmas

The following subsections describe the form, allowed places, and effect of every implementation-dependent pragma in UCS Ada. Unless noted in this section, UCS Ada implements all pragmas as described in Section 13 of the *UCS Ada Programming Reference Manual Volume 1*.

The following pragmas are allowed in an Ada program but the compiler ignores them. These pragmas have no effect on the compilation of the program or the program itself.

- MEMORY_SIZE
- OPTIMIZE
- STORAGE_UNIT
- SYSTEM_NAME

The only pragmas that are permitted to appear before a compilation unit (or a context clause preceding a compilation unit) are LIST and PAGE.

F.1.1. Standard Pragmas

UCS Ada supports the standard pragmas described in Section 13 of the *UCS Ada Programming Reference Manual Volume 1*. The following table identifies any characteristics of these pragmas that are specific to the UCS implementation of Ada.

Pragma	Characteristics
CONTROLLED	The compiler performs automatic storage reclamation whether or not a program specifies this pragma.
INLINE	<p>Pragma INLINE only turns on inline subprogram expansion when you specify the compiler keyword option EXPAND on the compiler call. Additionally, the EXPAND keyword option does not affect the program unless you specify the EXPAND option when you compile the calling subprogram and the procedure body.</p> <p>You can also turn on inline subprogram expansion without using pragma INLINE by using just the EXPAND option at compile time. In this case, inline expansion to increase execution efficiency only occurs on local subprograms that are called once.</p> <p>The compiler will never inline a subprogram that is called in a declarative part of a program unit.</p>

continued

continued

Pragma	Characteristics
INTERFACE	See Section for a description of this pragma.
PACK	<p>The following restrictions apply to this pragma:</p> <ul style="list-style-type: none"> • It is supported for an array only if any of the constraints on the array element or any of its subcomponents is static. If this is not the case, the compiler ignores the pragma. • It is not allowed for records
SUPPRESS	This pragma also supports the check identifier ALL_CHECKS. Specifying this option suppresses all of the run-time checks that can be performed on a compilation unit.

F.1.2. Nonstandard Pragas

This section describes pragmas that are not defined by the Ada standard, but that are available in UCS Ada.

Pragma INTERFACE_NAME

See Section for a description of this pragma.

Pragma IMPROVE

This pragma provides the capability to suppress certain implicit (compiler-generated) components in record types. This improves the following for records:

- Space usage
- Processing time

Formats

```
pragma IMPROVE(TIME,record_type);
pragma IMPROVE(SPACE,record_type);
```

where *record_type* denotes which record type should have the chosen representation. Pragma IMPROVE is ignored if the second argument is not a record type.

If TIME is specified, the compiler inserts all of the necessary implicit record components needed for the record type. These components are as follows (see F.2.2 and F.4.10):

- RECORD_SIZE
- VARIANT_INDEX

- `ARRAY_DESCRIPTOR`
- `RECORD_DESCRIPTOR`

The `TIME` format is the default case if `pragma IMPROVE` is not explicitly specified for a record type.

When you use the `SPACE` format, the compiler inserts a `VARIANT_INDEX` or a `RECORD_SIZE` component only if the component appears in a record representation clause that applies to the record type. You can use a record representation clause in this way to keep one implicit component while suppressing the other. The use of this pragma results in a smaller record storage area. However, this space savings causes slower program execution because the compiler must generate and call additional subprograms to compensate for the reduced amount of information that is available to the compiler.

A pragma `IMPROVE` statement that applies to a given record type can occur anywhere that a representation clause is allowed for the type. A program can only specify one pragma `IMPROVE` statement for a given record type. The compiler ignores the second and subsequent occurrences of the pragma for the record type.

Pragma `INDENT`

Section describes the use of the `INDENT` pragma.

Pragma `NON_ADA_ACCESS`

This pragma is used on an access type. It specifies that an object pointed to by an access object of this type may not be in the standard bank used for Ada allocations (the Ada heap bank).

Format

```
pragma NON_ADA_ACCESS(access_type);
```

where *access_type* is an Ada access type.

This pragma could be used on a pointer that is set outside of Ada code (for example, a parameter passed to UCS Pascal that is an OUT access object).

When you specify `pragma NON_ADA_ACCESS` for an access type, the internal format of an access object uses a 36-bit extended mode virtual address (in the L-BDI-offset format). The internal format of an access object whose type does not specify this pragma is a 36-bit Ada heap bank offset.

If a user is performing `UNCHECKED_CONVERSION` between objects of type `access` and type `SYSTEM.ADDRESS`, the access type should be declared using pragma `NON_ADA_ACCESS` because an object of type `SYSTEM.ADDRESS` contains a 36-bit virtual address.

Generally, do not use the pragma `NON_ADA_ACCESS` on an access type when the type being accessed and the access type itself are both unconstrained (as an example, type `ACC_STRING` is `STRING`;). The exception to this rule is if the access object is initialized through a `NEW` statement. The reason for this is that the compiler assumes that a descriptor for the unconstrained object is found in the storage that precedes the object.

F.2. Attributes

This subsection lists the name and type of every implementation-dependent attribute in UCS Ada. The following types of attributes are described in the following subsections:

- Standard
- Non-Standard

F.2.1. Standard Attributes

UCS Ada implements all standard attributes in accordance with the Ada standard.

See F.9.2 for the values given to the attributes for floating-point types.

F.2.2. Nonstandard Attributes

UCS Ada implements the following attributes that are not defined in the Ada standard:

- Record representation clause attributes
- Other nonstandard attributes

Record Representation Clause Attributes

The compiler creates special record components for certain record type definitions. Such record components are implementation-dependent; the compiler uses them to improve the quality of the generated code for certain operations on the record types. The criteria the compiler uses to create these components are implementation-dependent.

The record representation clause attributes

- Are only allowed in record representation specifications
- Are implicitly inserted by the compiler, if not specified
- Are not needed if used in application programs involving only Ada code
- Can be used for interlanguage calls passing records as parameters

The following is a list of the attributes that you can refer to in record representation clauses:

Implementation-Dependent Language Features

Attribute	Description
ARRAY_DESCRIPTOR	C'ARRAY_DESCRIPTOR is allowed for a prefix C that denotes a record component of an array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the compiler to store information on subtypes of components that depend on discriminants.
OFFSET	<p>C'OFFSET is allowed for a prefix C that denotes a record component of an array or record type whose constraints are not static. This attribute refers to the record component introduced by the compiler to store the offset of component C from the start of the record object.</p> <p>This component exists for components that have dynamic constraints, except for the first such component.</p>
RECORD_DESCRIPTOR	C'RECORD_DESCRIPTOR is allowed for a prefix C that denotes a record component of a record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the compiler to store information on subtypes of components that depend on discriminants.
RECORD_SIZE	<p>T'RECORD_SIZE is allowed for a prefix T that denotes a record type. This attribute refers to the record component introduced by the compiler to store the size of the record object.</p> <p>This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants (unless a pragma IMPROVE statement applies to the record type).</p>
VARIANT_INDEX	<p>T'VARIANT_INDEX is allowed for a prefix T that denotes a record type. This attribute refers to the record component introduced by the compiler to assist in the efficient implementation of discriminant checks.</p> <p>This component exists for objects of a record type with a variant part (unless a pragma IMPROVE statement applies to the record type).</p>

The compiler issues an error message if you refer to an implementation-dependent component that does not exist in the specified record. If the implementation-dependent component does exist, the compiler checks that the storage location specified in the record representation clause is compatible with the treatment of this component and the storage locations of other components. The compiler issues an error message if this check fails.

Implementation-Dependent Language Features

Other Nonstandard Attributes

In addition to the attributes used in record representation clauses, UCS Ada defines the following implementation-dependent attributes:

Attribute	Description
EXCEPTION_CODE	E'EXCEPTION_CODE applies to an exception name E. It returns the internal code associated with the exception. The value of this attribute is of the type <i>universal-integer</i> .
DESCRIPTOR_SIZE	T'DEScriptor_SIZE applies to a type or subtype T. It returns the size in bits of the descriptor associated with the object or type (this size is defined to be 0 if the type is not an array type or has no associated descriptor). The value of this attribute is of the type <i>universal-integer</i> .
IS_ARRAY	T'IS_ARRAY applies to a type or subtype T. It returns TRUE if the type is an array type, FALSE if it is not. The value of this attribute is of the predefined type BOOLEAN.

F.3. Package SYSTEM

This subsection presents the specification of package SYSTEM, which contains system-dependent configuration information. See the *UCS Ada Programming Reference Manual Volume 1* for more information about this package.

```
package SYSTEM is

  -- System Dependent Named Numbers

  MIN_INT      : constant := -(2**35 - 1) ;
  MAX_INT      : constant :=  2**35 - 1  ;
  MAX_DIGITS   : constant := 17 ;
  MAX_MANTISSA : constant := 35 ;
  FINE_DELTA   : constant := 2#1.0#E-35 ;
  TICK         : constant := 0.0002 ;

  type ADDRESS is range MIN_INT .. MAX_INT ;

  type NAME is (UCS) ;
  SYSTEM_NAME : constant NAME := NAME'FIRST ;

  STORAGE_UNIT : constant := 36 ;

  NUMBER_OF_BANKS : constant := 64 ;
  WORDS_IN_A_BANK : constant := 262_144 ;
  MEMORY_SIZE     : constant := NUMBER_OF_BANKS * WORDS_IN_A_BANK ;

  subtype PRIORITY is INTEGER range 1 .. 1 ;

  procedure ELIMINATE_NEGATIVE_ZERO (OBJECT : in out INTEGER);

end SYSTEM;
```

F.4. Representation Clauses

This subsection explains how the UCS Ada compiler represents and allocates objects and how it is possible to control this using representation clauses. The subsection also describes the applicable restrictions on representation clauses.

The representation of an object is closely connected with its type. For this reason, the discussion that follows addresses the representation of the following types and their corresponding objects:

- Enumeration
- Character
- Boolean
- Integer
- Floating point
- Fixed point
- Access
- Task
- Array
- Record

Except in the case of array and record types (called composites), the description of each type is independent of the others. To understand the concepts of array and record types, it is first necessary to understand their components.

Apart from implementation-defined pragmas, Ada provides three means to control the size of objects:

- A (predefined) pragma PACK statement when the object is an array
- A record representation clause, when the object is a record or a record component
- A size specification

For each class of types, the effect of a size specification is described. Interaction between size specifications, packing and record representation clauses is described under array and record types.

UCS Ada does not support size representation clauses on types derived from private types when the derived type is declared outside the private part of the defined package.

The representation of data types are described in terms of their logical and default sizes, and their physical size and alignment as defined below:

Characteristic	Description
Logical size	The minimum number of bits acceptable to hold the value of the type. This is also called the minimum size.
Default size	The smallest appropriate data size manipulated directly by the target for the type in question.
Physical size	The memory size actually allocated to hold the object. Usually this will be one of the sizes above and cannot be smaller than the logical size.
Physical alignment	The alignment in memory of the allocated storage with respect to byte or word boundaries.

For scalars, the entire storage allocated to an object (the physical size) is loaded when the program accesses the object. Bits within the physical size but not within the logical size are zero for unsigned quantities, or sign extended for signed quantities.

The design of UCS Ada limits the default representation of data types to that which the hardware directly and efficiently supports. However, representation clauses provide user control of data representation at the bit level. These representations are often not handled as efficiently as the defaults.

Types and Subtypes

By default, subtypes have the same internal representation as their base type. It is possible to represent a subtype with a range constraint that is both

- Contained in but not equal to the range of its base type
- Contained in a smaller amount of storage than the base type

For example, a program can map a subrange of INTEGER onto a quarter word (9 bits) if the bounds of the constraint fall within the range -255 .. 255. UCS Ada does not adopt such representations by default.

Derived Types

By default, UCS Ada maps derived types the same as their base types.

F.4.1. Enumeration Types

The following information describes the UCS Ada representation of enumeration types.

Internal Codes of Enumeration Literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Thus, for an enumeration type with n elements, the internal codes are the integers, 0, 1, 2, 3, ..., $n-1$.

A program can provide an enumeration representation clause to specify the value of each internal code as described in the discussion on enumeration representation clauses in the *UCS Ada Programming Reference Manual*. The UCS Ada Compiler fully implements enumeration representation clauses.

As internal codes must be machine integers, enumeration representation clauses must provide internal codes in the range between $-2^{35}-1$ and $2^{35}-1$.

Encoding of Enumeration Values

By default, enumeration types are represented as 36-bit unsigned quantities. Their logical size is the minimum number of bits needed to hold the value of the upper bound. The value of an enumeration element is its position number as defined in the discussion on enumeration values in the *UCS Ada Programming Reference Manual Volume 1*. A program processed by the UCS Ada compiler always represents an enumeration value by its internal code.

Minimum Logical Size of an Enumeration Subtype

The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range, its minimum size is 1. Else, the minimum size is calculated as follows:

If ...	Then the minimum size is ...
$m \geq 0$	the smallest positive integer such that $M \leq 2^{**}L-1$.
$m < 0$	the smallest positive integer such that $-2^{**}(L-1)-1 \leq m$ and $M \leq 2^{**}(L-1)-1$.

Legend

m is the lower bound of the subtype
 M is the upper bound of the subtype
 L is the minimum size in bits

Example

```

type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X..X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of the static type mark
-- BLACK_AND_WHITE).
```

Size of an Enumeration Subtype

The default size of all enumeration types is 36 bits. UCS Ada does not support enumeration types containing more than $2^{35}-1$ elements. The default alignment is word alignment.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

Example

```

type ENUM_TYPE1 is (A,B,C,D,E,F,G,H);
for ENUM_TYPE1'SIZE use 3;
-- The size of type ENUM_TYPE1 is three bits.

type ARRAY_TYPE1 is array(1 .. 100) of ENUM_TYPE1;
ARRAY1: ARRAY_TYPE1;
-- The size of ARRAY1 is 300 bits.
-- Without the 'SIZE clause for ENUM_TYPE1,
-- the size of ARRAY1 would be 100 words
```

UCS Ada fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 36 bits.

Size of the Objects of an Enumeration Subtype

An object of an enumeration subtype has the same size as its subtype provided its size is not constrained by the following:

- A record component clause
- A PACK pragma

User-Specified Representations

The compiler only accepts a user-specified enumeration representation clause if it has the following properties:

- Has a range between $2^{35}-1$ and $-(2^{35}-1)$.
- Preserves the ordering relationship on the type (Ada language requirement)

Note, in particular, that UCS Ada allows negative values.

The logical size of an enumeration type to which an enumeration representation clause applies is the logical size of an integer type whose bounds are the lower and upper values specified by the user. Hence, the enumeration type is mapped as this integer type is mapped.

A user-specified length clause is only accepted if it specifies n bits, where n is no larger than 36 and no smaller than the logical size of the enumeration type. Note that a length clause does not change the alignment of a data item. You can, however, control the alignment of data within arrays and records by using pragma PACK and record representation clauses, respectively.

F.4.2. Character Types

Character types are enumeration types containing at least one character literal (see the *UCS Ada Programming Reference Manual Volume 1*). However, storage allocation for the predefined type character and subtypes of character are somewhat different than for enumeration types. This section describes the differences from enumeration types.

Derived types of the predefined type character follow the normal rules for enumeration types. The following table lists the character type attributes and their descriptions.

Attribute	Description
Type	Default physical size: 9 bits
Container	Default size: 36 bits (that is, a word is allocated for a character object, even though only 9 bits in the word are used).
Justification	<p>The justification of a character object in a word is as follows:</p> <ul style="list-style-type: none"> • A record component of type character is left justified. Therefore, by default, a character object goes in the leftmost 9 bits (also known as Q1) of the word. The rest of the word is not used (unless a record representation clause used for the record type causes some other object to be placed there). • A character object that is not a record component is right justified. Therefore, a standalone character scalar goes in the rightmost 9 bits (also known as Q4) of the word.

By default, allocation of an array of type character is 1 word per array element. However, there are ways to get around this type of allocation. For example, to get an array of type character with four elements per word (that is, 1 array element per byte), one of the following could be done:

- Specify pragma PACK for the array type. For example:

```
type ARR1_TYPE is array(1..100) of CHARACTER;
pragma PACK(ARR1_TYPE);
```

Note that the predefined type string is defined as an array of predefined type character using pragma PACK, so a string object has four characters per word.

- Specify a size clause of nine bits for a derived character type. For example:

```
type CHAR1_TYPE is new CHARACTER;
for CHAR1_TYPE'SIZE use 9;
type ARR2_TYPE is array(1..100) of CHAR1_TYPE;
```

If a size specification is used for a character type, the length must be greater than or equal to the minimum size (normally seven bits), and less than or equal to 36 bits.

F.4.3. Boolean Types

Boolean types are enumeration types. However, storage allocation for the predefined type `boolean` and subtypes of `boolean` are somewhat different than for enumeration types. This section describes the differences from enumeration types.

Derived type of the predefined type `boolean` follow the normal rules for enumeration types. The following table lists the boolean type attributes and their descriptions.

Attribute	Description
Physical Size	Default physical size: 9 bits
Container Size	Default container size: 36 bits (that is, a word is allocated for a boolean object, even though only 9 bits in the word are used). In the 9 bits of a boolean object, the upper eight bits always contain zero, with the low bit being either zero (FALSE) or one (TRUE).
Justification	<p>The justification of a boolean object in a word is as follows:</p> <ul style="list-style-type: none"> • A record component of type <code>boolean</code> is left justified. Therefore, by default, a boolean object goes in the leftmost 9 bits (also known as Q1) of the word. The rest of the word is not used (unless a record representation clause used for the record type causes some other object to be placed there). • A boolean object that is not a record component is right justified. Therefore, a standalone boolean scalar goes in the rightmost 9 bits (also known as Q4) of the word.

By default, allocation of an array of type `boolean` is 1 word per array element. However, there are ways to get around this type of allocation. The following are examples.

- To get an array of type `boolean` with 36 elements per word (that is, 1 array element per bit), specify `pragma PACK` for the array type. For example:

```
type ARR3_TYPE is array(1..200) of BOOLEAN;
pragma PACK(ARR3_TYPE);
```

- To get an array of type `boolean` with 4 elements per word (that is, 1 element per byte), use a size clause specifying 9 bits for a derived boolean type. For example:

```
type BOOL1_TYPE is new BOOLEAN;
for BOOL1_TYPE'SIZE use 9;
type ARR4_TYPE is array(1..50) of BOOL1_TYPE;
```

If a size specification is used for a boolean type, the length must be greater than or equal to the minimum size (normally one bit), and less than or equal to 36 bits.

F.4.4. Integer Types

The following information describes the UCS Ada representation of integer types.

Encoding Integer Values

The Unisys 1100/2200 architecture supports one's complement integer values. The full set of arithmetic operations is available on full-word integer values only. Thus, there is one predefined integer type, `INTEGER`, with the following attributes:

```
INTEGER'FIRST  -235-1    (-34_359_738_367)
INTEGER'LAST   235-1    (+34_359_738_367)
```

By default, the physical size of an integer object is 36 bits and it is word aligned.

Minimum Size of an Integer Subtype

The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form. That is, the representation is in an unbiased form that includes a sign bit only if the range of the subtype includes negative values.

If a static subtype has a null range, the minimum size is 1. Otherwise, the minimum size is calculated as follows:

If ...	Then the minimum size is ...
$m \geq 0$	the smallest positive integer such that $M \leq 2^{L-1}$.
$m < 0$	the smallest positive integer such that $-2^{L-1} \leq m$ and $M \leq 2^{L-1}$.

Legend

m is the lower bound of the subtype

M is the upper bound of the subtype

L is the minimum size in bits

Example

```
subtype S is INTEGER range 0..7;
-- The minimum size of S is 3 bits.

subtype D is S range X .. Y;
```

Implementation-Dependent Language Features

- Assuming that X and Y are not static, the minimum size of
- D is 3 bits (the same as the minimum size of its type mark S).

Size of an Integer Subtype

The size of the predefined INTEGER type is 36 bits by default.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is 36 bits.

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

Example

```
type INT1_TYPE is range 1 .. 511;
for INT1_TYPE'SIZE use 9;
-- INT1_TYPE is an integer type, but its size is
-- 9 bits because of the SIZE clause.
```

```
subtype INT2TYPE is INT1_TYPE range 0 .. 255;
-- The size of INT2_TYPE is 9 bits because the
-- size of its base type is 9 bits.
```

UCS Ada fully implements size specifications. Nevertheless, as integers are implemented using one-word integers, the specified length cannot be greater than 36 bits.

Size of Objects of an Integer Subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

User-Specified Representation

The UCS compiler accepts a user-specified length clause only if it specifies n bits, where n is no larger than 36 and no smaller than the logical size of the integer type. Note that a length clause does not change the alignment of a data item. The alignment of data within arrays and records can, however, be controlled by use of the PACK pragma and record representation clauses.

F.4.5. Floating-Point Types

The following information describes the UCS Ada representation of integer types.

Encoding Floating-Point Values

UCS Ada directly maps two predefined floating types, `FLOAT` and `LONG_FLOAT`, onto the Unisys 1100/2200 architecture to support floating-point formats. The formats are as follows:

```

type FLOAT is digits 7
  range -(2.0**127*(1.0-2.0**(-27))) ..
    2.0**127*(1.0-2.0**(-27));
-- range approximately -(1.70*10.0**38) .. 1.70*10.0**38;

type LONG_FLOAT is digits 17
  range -(2.0**1023*(1.0-2.0**(-60))) ..
    2.0**1023*(1.0-2.0**(-60));
-- range approximately -(8.99*10.0**307) .. 8.99*10.0**307;

```

Other `FLOAT` and `LONG_FLOAT` characteristics are listed in F.9.2.

By default, UCS Ada internally represents floating-point types as one of the predefined representations described above. Which predefined representation is chosen depends on the number of digits specified in the `DIGITS` clause in a type specification, as described below:

Default Size	Digits
36 bits (<code>FLOAT</code>)	1..7
72 bits (<code>LONG_FLOAT</code>)	8..17

Floating-point types with more than 17 digits are not supported.

By default, the physical (allocated) size for a floating-point object is its default size as above and it is word aligned.

Minimum Size of a Floating-Point Subtype

The minimum size of a floating-point subtype is one of the following:

- 36 bits if its base type is `FLOAT` or a type derived from `FLOAT`
- 72 bits if its base type is `LONG_FLOAT` or a type derived from `LONG_FLOAT`

Size of a Floating-Point Subtype

The sizes of the predefined floating-point types `FLOAT` and `LONG_FLOAT` are respectively 36 and 72 bits.

The size of a floating-point type or the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating-point type or first-named subtype by using a size specification is its usual size (36 or 72 bits).

Size of Objects of a Floating-Point Subtype

An object of a floating-point subtype has the same size as its subtype.

User-Specified Representation

UCS Ada only accepts a user-specified length clause if it specifies the default size (36 or 72 bits) as previously described.

F.4.6. Fixed-Point Types

The following information describes the UCS Ada representation of fixed-point types.

'SMALL Specification of a Fixed-Point Type

If no specification of 'SMALL applies to a fixed-point type, then the value of 'SMALL is determined by the value of delta as defined by the *UCS Ada Programming Reference Manual Volume 1*.

Restriction

T'SMALL must have a value that is a power of 2. The value of the exponent must be between -1059 and 988.

Encoding of Fixed-Point Values

The UCS Ada compiler manages fixed-point types as the product of a signed mantissa and the constant 'SMALL. UCS Ada implements the signed mantissa as a signed integer. 'SMALL is a compile-time quantity that is the largest power of 2 that is less than the delta specified in the declaration of the type.

Thus, UCS Ada implements fixed point types as one's complement values. A fixed point object must fit in 36 bits or less.

By default, UCS Ada implements fixed-point numbers as 36-bit, one's-complement values and aligns them on a word boundary. If we define MANTISSA as the smallest number such that

$$2^{**}MANTISSA \geq \max(\text{abs}(\text{upper_bound}), \text{abs}(\text{lower_bound})) / \text{SMALL}$$

then the logical size of a fixed point-type is MANTISSA + 1 and hence, fixed-point types with MANTISSA > 35 are not supported.

A value V of a fixed-point subtype F is represented as the integer

$$V / F'BASE'SMALL$$

Example

```
type FIX1_TYPE is delta 0.01 range 0.0 .. 10.0;
-- The value of FIX1_TYPE'SMALL is 2**(-7),
-- which is the largest power of 2 that is less than 0.01
f1: FIX1_TYPE := 0.5;
-- The value 0.5 is represented as the integer
-- 0.5 / (2**(-7)), or 64
```

Minimum Size of a Fixed-Point Subtype

The minimum size of a fixed-point subtype is the minimum number of bits that is necessary for representing the values of the range of the subtype using the 'SMALL' value of the base type. That is, UCS Ada represents the minimum size in an unbiased form that includes a sign bit only if the range of the subtype includes negative values.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the lower and upper bounds of the subtype, if i and I are the integer representations of m and M , the smallest and greatest model numbers of the base type such that $s < m$ and $M < S$, then the minimum size L is determined as follows. For $i \geq 0$, L is the smallest positive integer such that $I \leq 2^{L-1}$. For $i < 0$, L is the smallest positive integer such that $-2^{L-1} \leq i$ and $I \leq 2^{L-1}$.

Example

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of D is 7 bits
-- (the same as the minimum size of its type mark S).
```

Size of a Fixed-Point Subtype

When no size specification is applied to a fixed-point type or to its first-named subtype, its size and the size of any of its subtypes is 36 bits.

When a size specification is applied to a fixed point-type, this fixed-point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first-named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the subtype to which it applies.

Example

```
type F is delta 2.0 range 0.0 .. 500.0;
for F'SIZE use 9;
-- As noted above, the minimum size of F is 8 bits.
-- The actual size of F is 9 bits because of the SIZE clause.
-- Without the SIZE clause, the size of F would be the default 36 bits.
```

UCS Ada implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 36 bits.

Size of the Objects of a Fixed-Point Subtype

Provided its size is not constrained by a record component clause or a PACK pragma, an object of a fixed-point type has the same size as its subtype.

User Specified Representations

UCS Ada does not support representation clauses that specify 'SMALL to be other than a power of 2.

The compiler only accepts a user-specified length clause if it specifies n bits, where n is no smaller than the logical size or no larger than 36.

F.4.7. Composite Types

UCS Ada implements all composite (array and record) types by storing them contiguously in memory for easier access, input, and output. By default, the representation of an element or component is the default representation for its subtype. That is, the default representation of an object is the same whether it is an object by itself or a part of an enclosing object. If the elements or components are themselves arrays or records it is impossible to change their representation.

F.4.8. Array Types

UCS Ada implements array objects by storing them contiguously in memory. All the components have the same size. A gap can exist between two consecutive components (and after the last one). All the gaps have the same size.

Array Components

If the array is not packed (that is, pragma PACK is not specified for the array type), the size of the components is the size of the subtype of the components.

Example

```
type A is array (1..8) of BOOLEAN;
-- The size of a component of A is the container size of
-- type BOOLEAN (36 bits)
-- The size of an array of type A is 8 words.

type DECIMAL_DIGIT is range 0..9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
    array (INTEGER range 0) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented in
-- 4 bits.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components.

Example

```
type A is array (1..8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN
-- which is one bit
type DECIMAL_DIGIT is range 0..9;
type BINARY_CODED_DECIMAL is
    array (INTEGER range 0) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 36 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component will be represented in
-- 4 bits.
```

Packing the array has no effect on the size of the components when the components are records or arrays.

Gaps

If the components are records or arrays, no size specification applies to the subtype of the components, and the array is not packed, then the compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype.

If no size specification applies to the subtype of the component, or if the array is packed, no gaps are inserted.

Examples of arrays of scalars

ase 1

```
type INT1_TYPE is range 0 .. 100;
type ARR1_TYPE is array(1..10) of INT1_TYPE;
-- Each component of an array of type ARR1_TYPE is one word
-- (36 bits), so every component starts on a word boundary.
-- An array of type ARR1_TYPE is ten words (360 bits).
```

Case 2

```
type ARR2_TYPE is array(1..10) of INT1_TYPE;
pragma PACK(ARR2_TYPE);
-- Each component of an array of type ARR2_TYPE is seven bits,
-- which is the minimum size of type INT1_TYPE.
-- An array of type ARR2_TYPE is 70 bits.
```

Case 3

```
type INT2_TYPE is new INT1_TYPE;
for INT2_TYPE'SIZE use 9;
type ARR3_TYPE is array(1..10) of INT2_TYPE;
-- Each component of an array of type ARR3_TYPE
-- is one byte (nine bits), because of the SIZE clause.
-- An array of type ARR3_TYPE is ten bytes (90 bits).
```

Gaps in arrays of records

Record components are allocated on full word boundaries, unless a record representation clause appears for the record type. For example, if a record component of type STRING(1..2) is declared, it is allocated a full word in the record (if no record representation clause appears), even though the component occupies only two bytes of the word.

If a record is declared with one or more components that do not have sizes that are multiples of 36 bits, then the only way to eliminate gaps in the record is to use a record representation clause.

If an array of such a record type (with a record representation clause) is declared, then no gaps are inserted in the array if one of the following conditions apply:

- a SIZE specification applies to the record type
- a pragma PACK applies to the array type.

Examples

Case 1

```
type R is
  record
    K: INTEGER;
    B: STRING(1..2);
  end record;

type A1 is array(1..10) of R;
-- Each component of A1 is two words (72 bits),
-- since no record representation clause appears for R.
-- A gap of two bytes is inserted after each component B
-- to force the word alignment of each array component.
-- The size of an array of type A1 is 720 bits (10 words).
```

Case 2

```
type A2 is array(1..10) of R;
pragma PACK(A2);
-- As with case 1, each component of A2 is two words (72 bits).
-- The pragma PACK has no effect, since there is no
-- record representation clause for R.
-- The size of an array of type A2 is 720 bits (10 words).
```

Case 3

```
type NR is new R;

for NR use
  record
    K at 0 range 0..35;
    B at 1 range 0..17;
  end record;

type A3 is array(1..10) of NR;
```

```
-- Each component of A3 is 54 bits (6 bytes).
-- A gap of two bytes is inserted after each component B
-- to force the word alignment of each array component,
-- since no SIZE clause appears for NR,
-- and no pragma PACK appears for A3.
-- The size of an array of type A3 is 720 bits (10 words).
```

Case 4

```
type A4 is array(1..10) of NR;
pragma PACK(A4);
-- Each component of A4 is 54 bits.
-- Gaps are eliminated, because of the record representation
-- clause for NR and the pragma PACK.
-- The size of an array of type A4 is 540 bits (15 words).
```

Case 5

```
type NR1 is new NR;
for NR1'SIZE use 54;
type A5 is array(1..10) of NR1;
pragma PACK(A5);
-- Each component of A5 is 54 bits.
-- Gaps are eliminated, because of the record representation
-- clause for NR and the SIZE clause.
-- The size of an array of type A5 is 540 bits (15 words).
```

Size of an Array Subtype

By default, the elements of an array are allocated by rows, each in its default representation and on its default alignment. The default alignment of an array as a whole is the default alignment for its element type.

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is constrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time if

- It has nonstatic constraints or is an unconstrained array type with nonstatic index subtypes (because the number of components can then only be determined at run time)
- The components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time)

Implementation-Dependent Language Features

As has been indicated above, the effect of pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, the UCS Ada compiler fully implements array packing.

Size of an Object of an Array Subtype

The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment of an Array Subtype

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

User-Specified Representations

The only size that can be specified for an array type or first-named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

F.4.9. String Types

A string type is treated as a packed array of predefined type CHARACTER. Therefore, strings are mapped four characters to a word.

F.4.10. Record Types

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in the *UCS Ada Programming Reference Manual Volume 1*, Section 13.4. In UCS Ada, there is no restriction on the

position specified for a component of a record. Bits within a storage unit are numbered from 0 to 35, with the most significant (leftmost) bit numbered 0. The range of bits specified in a component clause may extend into following storage units. If a component is not a record or an array, its size can be any size from the minimum sized to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype.

Example

```

type REC1_TYPE is
  record
    F1: LONG_FLOAT;
    B1, B2: BOOLEAN;
  end record;

for REC1_TYPE use -- record representation clause
  record
    F1 at 0 range 0 .. 71; -- Words 0-1
    B1 at 2 range 0 .. 8;  -- Bits 0-8 (first byte) of word 2
    B2 at 2 range 9 .. 17; -- Bits 9-17 (second byte) of word 2
  end record;

-- If no record representation clause appeared for REC1_TYPE,
-- F1 would be in words 0-1 and
-- B1 would be in the first byte of word 2.
-- However, B2 would appear in the first byte of word 3,
-- since all record components would be word aligned.

```

A record representation clause need not specify the position and the size for every component. If no component clause applies to a component of a record, its size is the size of its subtype. The compiler chooses its position so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of the alignment of the component subtype. Moreover, the compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects. Note, in particular, that variant parts are overlaid when possible and that holes within previously laid out variant parts are made use of whenever possible.

Without a record representation clause, each record component normally begins on a word boundary.

Because of these optimizations, there is not necessarily a connection between the order of the components in a record type declaration and the positions chosen by the compiler of the components in a record object.

Pragma PACK has no further effect on records. The UCS Ada compiler always optimizes the layout of records as described above.

Indirect Components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:

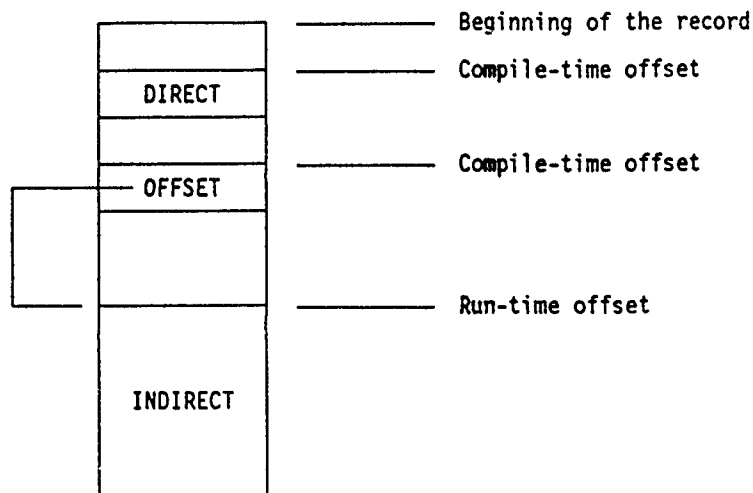


Figure F-1. Direct and Indirect Components

If a record component is a record or an array, the size of its subtype can be evaluated at run time and can even depend on the discriminants of the record. UCS Ada calls these components dynamic components.

Example

```
type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range  $\diamond$ ) of INTEGER;

type GRAPH (L : NATURAL) is
  record
    X : SERIES(1 .. L); -- The size of X depends on L
    Y : SERIES(1 .. L); -- The size of Y depends on L
  end record;

Q : POSITIVE;

type PICTURE (N : NATURAL; D : DEVICE) is
  record
    F : GRAPH(N); -- The size of F depends on N
    S : GRAPH(Q); -- The size of S depends on Q
    case D is
      when SCREEN =>
        C : COLOR;
      when PRINTER =>
        null;
    end case;
  end record;
```

Any component placed after a dynamic component has an offset that cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the compiler groups dynamic components together and places them at the end of the record as follows:

Implementation-Dependent Language Features

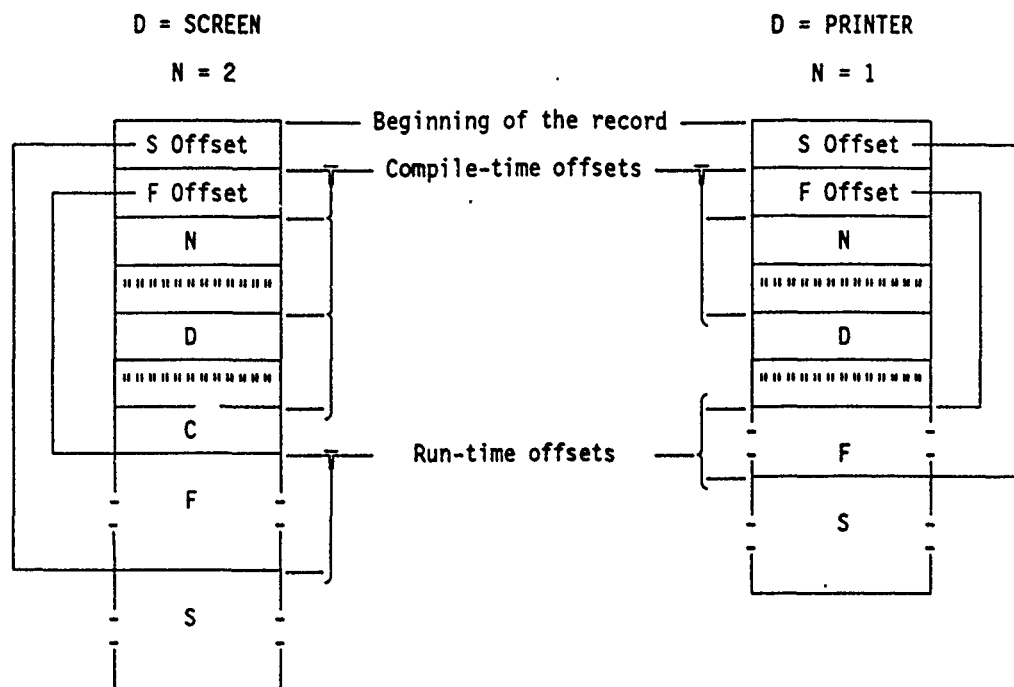


Figure F-2. Record Type PICTURE: F and S Are Placed at the End of the Record

Due to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect. If there are dynamic components in a component list that is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time.

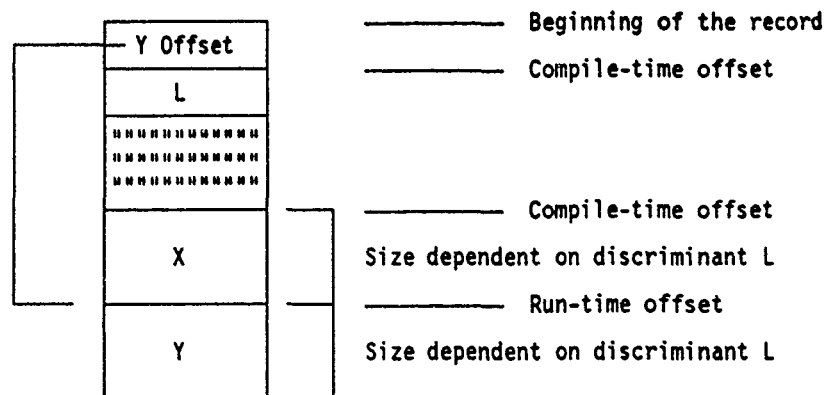


Figure F-3. Record Type GRAPH: The Dynamic Component X Is a Direct Component

For every dynamic component of a record (except the first), a static offset pointer is created to contain the word or bit displacement of the dynamic component from the start of the record. If these bounds are dynamic, for example in the case of the offset pointer of the second of two dynamic sized components, the corresponding offset pointer is always one word long.

If *C* is the name of an indirect component, the offset of this component can be denoted in a component clause by the implementation generated name *C'OFFSET*.

Implicit Components

In some circumstances, access to an object of a record type or to its components involves computing information that depends only on the discriminant values. To avoid recomputation (which would degrade performance) the compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or its components are accessed. This information is stored in special components called implicit components.

An implicit component can contain information that is used when the record object or several of its components are accessed. In this case the component is included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called *RECORD_SIZE* and the other *VARIANT_INDEX*.

On the other hand, an implicit component can be used to access a given record component. In that case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called an *ARRAY_DESCRIPTOR* or a *RECORD_DESCRIPTOR*.

RECORD_SIZE

This implicit component is created by the compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a *RECORD_SIZE* component may denote a number of bits or a number of storage units (words). In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size that cannot be expressed using storage units, the value designates a number of bits.

The implicit component *RECORD_SIZE* must be large enough to store the maximum size of any value of the record type. The compiler evaluates an upper bound *MS* of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. *MS*.

If *R* is the name of the record type, this implicit component can be denoted in a component clause by the implementation-generated name *R*'*RECORD_SIZE*.

This allows you control over the position of the implicit component in the record.

VARIANT_INDEX

This implicit component is created by the compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists in variant parts that themselves do not contain a variant part are numbered. These numbers are the possible values of the implicit component *VARIANT_INDEX*.

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);
```

```
type DESCRIPTION (KIND : VEHICLE := CAR) is
  record
    SPEED : INTEGER;
    case KIND is
      when AIRCRAFT | CAR =>
        WHEELS : INTEGER;
        case KIND is
          when AIRCRAFT => -- 1
            WINGSPAN : INTEGER;
          when others => -- 2
            null;
        end case;
      when BOAT => -- 3
        STEAM : BOOLEAN;
      when ROCKET => -- 4
        STAGES : INTEGER;
      end case;
    end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	{KIND, SPEED, WHEELS, WINGSPAN}
2	{KIND, SPEED, WHEELS}
3	{KIND, SPEED, STEAM}
4	{KIND, SPEED, STAGES}

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component `VARIANT_INDEX` must be large enough to store the number `V` of component lists that do not contain variant parts. The compiler treats this implicit component as having an anonymous integer type whose range is `1 .. V`.

If `R` is the name of the record type, this implicit component can be denoted in a component clause by the implementation-generated name `R'VARIANT_INDEX`.

This allows user control over the position of the implicit component in the record.

ARRAY_DESCRIPTOR

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind `ARRAY_DESCRIPTOR` is not described in this documentation.

The compiler treats an implicit component of the kind `ARRAY_DESCRIPTOR` as having an anonymous array type. If `C` is the name of the record component whose subtype is described by the array descriptor, this implicit component can be denoted in a component clause by the implementation generated name `C'ARRAY_DESCRIPTOR`. This attribute allows user control over the position of the implicit component in the record.

RECORD_DESCRIPTOR

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind `RECORD_DESCRIPTOR` is not described in this documentation.

The compiler treats an implicit component of the kind `RECORD_DESCRIPTOR` as having an anonymous array type. If `C` is the name of the record component whose

Implementation-Dependent Language Features

subtype is described by the record descriptor, this implicit component can be denoted in a component clause by the implementation-generated name `C'RECORD_DESCRIPTOR`. This attribute allows user control over the position of the implicit component in the record.

Suppression of Implicit Components

The UCS Ada implementation provides the capability of suppressing the implicit components `RECORD_SIZE` and `VARIANT_INDEX` from a record type. You can accomplish this by using an implementation-defined pragma called `IMPROVE` (see F.1.2).

```
pragma IMPROVE ( TIME | SPACE , [ON =>] simple_name );
```

The first argument specifies whether `TIME` or `SPACE` is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If `TIME` is specified, the compiler inserts implicit components as described above. If on the other hand `SPACE` is specified, the compiler only inserts a `RECORD_SIZE` and `VARIANT_INDEX` if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

An `IMPROVE` pragma that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

Size of a Record Subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of words.

By default, the components of a record are allocated contiguously according to the physical size and alignment requirements of their types. Each component is allocated its default physical size at its default alignment. Note, however, that the order of allocation of components is not necessarily the lexical order of the record declaration, but is chosen by the compiler to honor optimization requirements mentioned above. Thus, the size of a record type as a whole depends on the physical size and alignment requirements of its individual components. By default, a record as a whole is aligned at a word boundary.

User-Specified Representations

The compiler ignores a user-specified pragma `PACK` applied to a record type. It accepts a user-specified length clause (using the `SIZE` attribute) only if it specifies the exact length of the record type.

A user record representation clause is acceptable only if, for each component, the storage specified is no smaller than the logical size of the type of the component and the clause specifies the correct layout including implicit components.

F.4.11. Access Types

The following table describes the UCS Ada attributes of access types.

Attribute	Description
Encoding of Access Values	<p>Access types represent the addresses of data allocated in the heap. The heap mechanism always allocates an integral number of words and data is word aligned. Thus, UCS Ada always represents access values as word values. If pragma <code>NON_ADA_ACCESS</code> (see F.1.2) is used on the access type, the format is one word containing an extended mode virtual address (in the format, <i>L-BDI-offset</i>). Otherwise, the format is a one-word offset into the Ada heap bank.</p> <p>The compiler uses the special value zero (8#000000000000#) to represent the null access value. This value is chosen because it can be easily tested and the initialization of access values can be performed efficiently</p>
Access Subtype	<p>Minimum size of an access value: 36 bits Size of an access subtype: 36 bits</p> <p>An object of an access subtype has the same size as its subtype. Thus, an object of an access subtype is always 36 bits.</p>
User-Specified Representations	The compiler accepts a user-specified length clause only if it specifies the default size of 36 bits.

F.4.12. Task Types

The following table describes the attributes of task types.

Attribute	Description
Encoding of Task Values	A task object is represented as a 1-word (36 bit) extended mode virtual address representing the address of the Task Control Block (TCB) for the task. (The user has no control over the contents of the TCB.)
Size of a Task Type	A size specification ('SIZE) for a task type has no effect. The only size that can be specified using such a length clause is 36 bits. Example task type T; for T'SIZE use 36;
Storage for a Task Type	If a storage size ('STORAGE_SIZE) is specified for a task type, then it represents the number of words allocated in the fixed stack for a task of that type. Example task type T; for T'STORAGE_SIZE use 1000; -- A task of type T gets allocated 1000 words -- of fixed stack space. -- A STORAGE_ERROR exception results if the allocated -- stack space is not large enough during task execution. The following binder commands <ul style="list-style-type: none">• TASK STACK INIT SIZE• TASK STACK MAX SIZE• TASK STACK INC SIZE can be used to control the fixed stack sizes for all non-environment tasks created during program execution (see 4.2.3).

F.5. Implementation-Generated Names

This subsection lists the conventions that UCS Ada follows when generating names that denote implementation-dependent components.

The UCS Ada compiler may add implicit components to record objects. These fields contain information about the variant and storage layout of the record. The implementation-generated names are the following nonstandard attributes:

- R'RECORD_SIZE
- R'VARIANT_INDEX
- C'RECORD_DESCRIPTOR
- C'ARRAY_DESCRIPTOR
- C'OFFSET

where:

R

is the prefix denoting a record type

C

is the prefix denoting a record component

You can only use these names in representation clauses. See F.2.2 for an explanation of their meanings.

The other nonstandard attribute names are as follows:

- T'IS_ARRAY
- T'DEScriptor_SIZE
- E'EXCEPTION_CODE

where:

T

is the prefix denoting a type or subtype

E

is the prefix denoting an exception name

See F.2.2 for a description of these attributes.

The nonstandard pragmas are as follows:

- INTERFACE_NAME
- IMPROVE
- INDENT
- NON_ADA_ACCESS

Implementation-Dependent Language Features

See F.1.2 for a description of these nonstandard pragmas.

The UCS Ada compiler reserves the following predefined packages. Do not recompile these packages.

- UNISYS_ADA_RUNTIME
- UNISYS_BASIC_IO

F.6. Address Clauses

This subsection describes the interpretation of expressions that appear in address clauses, including those for interrupts. See the *UCS Ada Programming Reference Manual Volume 1*, Section 13.5 for more information.

An address clause has the following format:

for simple_name use at simple_expression;

where:

simple_name

must be one of the following:

- The name of an object
- The name of a subprogram, package, or task unit
- The name of a single entry

simple_expression

must be of type `SYSTEM.ADDRESS`

UCS Ada permits an address clause only if *simple_name* is an object.

F.7. Unchecked Conversions

The Ada standard, as documented in the *UCS Ada Programming Reference Manual Volume 1*, requires that each implementation list its restrictions on unchecked conversion.

Unchecked conversions are allowed only between types that have the same value for their 'SIZE' attribute (no error, however, is detected if either the target or the source is a composite). UCS Ada does not allow unchecked conversions between unconstrained array types.

It is the responsibility of the programmer to determine if the desired effect of the unchecked conversion has been achieved.

F.8. Input/Output

This subsection lists all of the implementation-dependent characteristics of the input-output packages. See Section 14 of the *UCS Ada Programming Reference Manual Volume 1* for the guidelines that the Ada standard establishes.

F.8.1. Ada Files

UCS Ada implements the input/output packages as the Ada language defines them. See F.8.2 for the implementation dependencies of the input/output packages.

Correspondence with OS 1100 Files

Ada defines the following operations for controlling external files:

- Procedure CREATE
- Procedure OPEN
- Procedure CLOSE
- Procedure DELETE
- Procedure RESET
- Function MODE
- Function NAME
- Function FORM
- Function IS_OPEN

Each of the Ada constructs for files declares this set of file management operations. These constructs include the following:

Implementation-Dependent Language Features

File Construct	Description
Sequential	<p>A sequence of records that are transferred one after the other. In addition to those listed above, the generic package <code>SEQUENTIAL_IO</code> provides the following operations:</p> <ul style="list-style-type: none">• Procedure <code>READ</code>• Procedure <code>WRITE</code>• Function <code>END_OF_FILE</code> <p>See the <i>UCS Ada Programming Reference Manual Volume 1</i>, Section 14.2.2 for more information.</p>
Direct	<p>A set of consecutive equal-sized records occupying consecutive positions in a linear order that can be transferred to or from any record of the file at any given position in the file. In addition to those listed above, the generic package <code>DIRECT_IO</code> defines the following operations for direct files:</p> <ul style="list-style-type: none">• Procedure <code>READ</code>• Procedure <code>WRITE</code>• Procedure <code>SET_INDEX</code>• Additional functions <code>INDEX</code> and <code>SIZE</code> <p>See Section 14.2.4 of the <i>UCS Ada Programming Reference Manual Volume 1</i> for more information.</p>
Text	<p>Provides the facility to do input/output in human-readable form. The package <code>TEXT_IO</code> provides the facilities and operations allowed for text files. In addition to those listed above, the <code>TEXT_IO</code> package defines the following operations for text files:</p> <ul style="list-style-type: none">• <code>GET</code>• <code>PUT</code>• <code>GET_LINE</code>• <code>PUT_LINE</code>• <code>SET_INPUT</code>• <code>SET_OUTPUT</code> <p>The functions defined for text files are</p> <ul style="list-style-type: none">• <code>STANDARD_INPUT</code>• <code>STANDARD_OUTPUT</code>• <code>CURRENT_INPUT</code>• <code>CURRENT_OUTPUT</code> <p>See the <i>UCS Ada Programming Reference Manual Volume 1</i>, Section 14.3 for more information on these functions along with several other formatting operations available for text files.</p>

Note: *The standard input and the standard output files cannot be opened, closed, reset, or deleted. For information describing why, see the UCS Ada Programming Reference Manual Volume 1.*

NAME Parameter

The NAME parameter supplied to the Ada procedures CREATE or OPEN (see the *UCS Ada Programming Reference Manual Volume 1*) can represent any of the legal OS 1100 file names.

The syntax of the Ada NAME parameter is as follows:

file_name ::= *legal_1100_file_name*

The syntax of a legal OS 1100 file name as specified in the Ada NAME parameter is as follows:

legal_1100_file_name ::= [*qual**]*file*[(*cycle*)]*[.]*

where:

qual
is the OS 1100 qualifier

file
is the 1100 file name

cycle
is the file cycle number

If either the qualifier or the file name exceeds 12 characters, then a NAME_ERROR exception is raised. As indicated above, the qualifier and the file cycle are not mandatory components of the NAME parameter. If the qualifier is omitted, the system defaults to the qualifier of the current user or to the specified project-id. If the file cycle is omitted, it defaults to the latest cycle of the file specified.

FORM Parameter

The FORM parameter supplied to the Ada procedures CREATE or OPEN is a string defining system-dependent characteristics that can be associated with a sequential, direct, or text input-output file. Examples of these characteristics are physical organization and access rights.

The attributes for defining a FORM string are described in the following text.

Implementation-Dependent Language Features

Package Name: SEQUENTIAL_IO

Attribute Name	Type/Range	Default
Block_size	Positive/ $1..2^{18}-1$	1792
Record_size	Positive/ $1..2^{16}-1$	256
Append	Boolean	False
Prep_Factor	Positive/ $224 \cdot X^{\#}$	1792
Labels	Standard or Omitted	Standard
Disposition	Leave or Rewind	Leave

[#] X: An integer in the range $1 < X \leq 1170$

The following table describes each attribute.

Attribute	Description
Block_size	The block size in words. This attribute is the file buffer. Records are written into the block until no more records will fit. At this point the whole block is actually written to the file. You should only change this attribute for performance considerations. There is not a set size to use for all cases with the exception that the block size must be a multiple of 28 words and a multiple of the prep factor.
Record_size	The record size in words. This attribute should be the size in words of the type instantiated with the package, if the type is constrained. If the type is unconstrained, this attribute must be specified.
Append	Determines whether to open a file in update mode.
Prep_Factor	The number of words used for the prep factor. This attribute is related to the block size in that the block size is a multiple of the prep factor.
Labels	Specifies whether the standard label should be used or if the labels are to be omitted.
Disposition	Determines how the file is left when it is closed. If LEAVE is specified, when the file is closed, it is left at the position after the last READ or WRITE operation. If REWIND is specified, when the file is closed, it is positioned at the beginning.

Package Name: DIRECT_IO

Attribute Name	Type/Range	Default
Block_size	Positive/ $1..2^{18}-1$	1792
Record_size	Positive/ $1..2^{16}-1$	256
Append	Boolean	False
Max_Rec_Num	Positive/ $1..Max_Int$	10000
Skeletonize	Boolean	True
Shared	Boolean	False
Prep_Factor	Positive/ $224*X^{\#}$	1792
Labels	Standard or Omitted	Standard
Disposition	Leave or Rewind	Leave

$\#$ X = an integer in the range of $1 < X \leq 1170$.

The following table describes each of the above attributes.

Implementation-Dependent Language Features

Attribute	Description
Block_size	The block size in words. This attribute is the file buffer. Records are written into the block until no more records will fit. At this point the whole block is actually written to the file. You should only change this attribute for performance considerations. There is not a set size to use for all cases with the exception that the block size must be a multiple of 28 words and a multiple of the prep factor.
Record_size	The record size in words. This attribute should be the size in words of the type instantiated with the package, if the type is constrained. If the type is unconstrained, this attribute must be specified.
Append	Determines whether to open a file in update mode.
Max_Rec_Num	Specifies the number of records that can be written to for the direct file. This is the size in records of the file.
Skeletonize	Specifies whether the file is skeletonized or not. It is highly recommended that the file be skeletonized. The only case where the file could not be skeletonized is if there are no holes in the file and it can be guaranteed that no one will ever try to read a record that has never been written.
Shared	Specifies whether the file is shared by other applications.
Prep_Factor	The number of words used for the prep factor. This attribute is related to the block size in that the block size is a multiple of the prep factor.
Labels	Specifies whether the standard label should be used or if the labels are to be omitted.
Disposition	Determines how the file is left when it is closed. If LEAVE is specified, when the file is closed it is left at the position after the last READ or WRITE operation. If REWIND is specified, when the file is closed it positioned at the beginning.

If a `RECORD_SIZE` or `MAX_REC_NUM` is specified when the file is created, then those same values must be used when opening the file for reading.

Package Name: TEXT_IO

Attribute Name	Type/Range	Default
Block_size	Positive/1..2 ¹⁸ -1	1792
Record_size	Positive/1..2 ¹⁶ -1	256
Append	Boolean	False
Prep_Factor	Positive/224*X [#]	1792
Labels	Standard or Omitted	Standard
Disposition	Leave or Rewind	Leave
Symb_Type	SDF, Read, Alt_Read Print, Alt_Print Punch, Alt_Punch	SDF

[#] The range of this attribute is 224.. 2¹⁸-1 but the value must be a multiple of 224.

The following table describes each of the above attributes.

Implementation-Dependent Language Features

Attribute	Description																
Block_size	The block size in words. This attribute is the file buffer. Records are written into the block until no more records will fit. At this point the whole block is actually written to the file. You should only change this attribute for performance considerations. There is not a set size to use for all cases with the exception that the block size must be a multiple of 28 words and a multiple of the prep factor.																
Record_size	Specifies the maximum length of a text line in words (4 bytes/word). The exception USE_ERROR is raised if a program tries to set the line length greater than this value.																
Append	Determines whether to open a file in update mode.																
Prep_Factor	The number of words used for the prep factor. This attribute is related to the block size in that the block size is a multiple of the prep factor.																
Labels	Specifies whether the standard label should be used or if the labels are to be omitted.																
Disposition	Determines how the file is left when it is closed. If LEAVE is specified, when the file is closed it is left at the position after the last READ or WRITE operation. If REWIND is specified, when the file is closed it is positioned at the beginning.																
Symb_Type	<p>Specifies the symbiont type to which the text file is associated. This type can be one of the following:</p> <table> <tr> <th>Type</th><th>Description</th></tr> <tr> <td>SDF</td><td>A sequential text file that can be written to, reset, and then read from. This file may not be edited or printed (with SYM) on an 1100 printer. To print this file, a program must be written to convert this text file to a symbiont file that may be edited or printed (with SYM). See example below.</td></tr> <tr> <td>Read</td><td>Usually associated with the standard read device READ\$. This attribute is considered similar to the STD_IN file.</td></tr> <tr> <td>Alt_Read</td><td>Similar to the read device but is not associated with the standard read device READ\$.</td></tr> <tr> <td>Print</td><td>Usually associated with the standard print device PRINT\$. This attribute is considered similar to the STD_OUT file.</td></tr> <tr> <td>Alt_Print</td><td>Similar to the print device but is not associated with the standard print device PRINT\$.</td></tr> <tr> <td>Punch</td><td>Usually associated with the standard punch device PUNCH\$.</td></tr> <tr> <td>Alt_Punch</td><td>Similar to the punch device but is not associated with the standard punch device PUNCH\$.</td></tr> </table>	Type	Description	SDF	A sequential text file that can be written to, reset, and then read from. This file may not be edited or printed (with SYM) on an 1100 printer. To print this file, a program must be written to convert this text file to a symbiont file that may be edited or printed (with SYM). See example below.	Read	Usually associated with the standard read device READ\$. This attribute is considered similar to the STD_IN file.	Alt_Read	Similar to the read device but is not associated with the standard read device READ\$.	Print	Usually associated with the standard print device PRINT\$. This attribute is considered similar to the STD_OUT file.	Alt_Print	Similar to the print device but is not associated with the standard print device PRINT\$.	Punch	Usually associated with the standard punch device PUNCH\$.	Alt_Punch	Similar to the punch device but is not associated with the standard punch device PUNCH\$.
Type	Description																
SDF	A sequential text file that can be written to, reset, and then read from. This file may not be edited or printed (with SYM) on an 1100 printer. To print this file, a program must be written to convert this text file to a symbiont file that may be edited or printed (with SYM). See example below.																
Read	Usually associated with the standard read device READ\$. This attribute is considered similar to the STD_IN file.																
Alt_Read	Similar to the read device but is not associated with the standard read device READ\$.																
Print	Usually associated with the standard print device PRINT\$. This attribute is considered similar to the STD_OUT file.																
Alt_Print	Similar to the print device but is not associated with the standard print device PRINT\$.																
Punch	Usually associated with the standard punch device PUNCH\$.																
Alt_Punch	Similar to the punch device but is not associated with the standard punch device PUNCH\$.																

A null string for the FORM specifies the use of the default options of the implementation for the external file.

Example

The example procedure converts an SDF PCIOS text file into an SYMB file that may be SYMed to a printer. The SDF PCIOS text contains a FF control character that causes problems if used as input to other 1100/2200 processors. This program converts the FF control character into the print control images that the other 1100/2200 processors understand.

```

with TEXT_IO;
use TEXT_IO;

procedure CONVERT_SDF_TO_SYMB is

    -- This first file type is the text file that was used for regular
    -- TEXT_IO processing without a FORM parameter.
    SDF_FILE : FILE_TYPE;

    -- This is the file that may be SYM'ed to a standard 1100/2200 printer
    -- or may be used in an editor.
    SYMB_FILE : FILE_TYPE;

    -- This is the variable that will be read from the SDF file and 'PUT'
    -- into the SYMB file.
    CHAR : CHARACTER := ' ';

begin -- CONVERT_SDF_TO_SYMB

    -- First open the SDF text file for input.
    OPEN( FILE => SDF_FILE,
          MODE => IN_FILE,
          NAME => "MYTEXTFILE" );

    -- Now create a SYMB text file to write the information from the SDF
    -- file.
    CREATE( FILE => SYMB_FILE,
            MODE => OUT_FILE,
            NAME => "SYMBFILE",
            FORM => "SYMB_TYPE => ALT_PRINT" );

    -- Get the first character in the file.
    GET(SDF_FILE,CHAR);

    -- Process the entire SDF_FILE until EOF is reached.
    while not END_OF_FILE(SDF_FILE) loop

        -- If the EOP is reached in the file, output a new page.
        if END_OF_PAGE(SDF_FILE) then
            NEW_PAGE(SYMB_FILE);
        end if;
    end loop;
end CONVERT_SDF_TO_SYMB;
```

Implementation-Dependent Language Features

```
-- If the EOL is reached in the file, output a new line.
elsif END_OF_LINE(SDF_FILE) then
    NEW_LINE(SYMB_FILE);

-- Output the character to the SYMB file.
else
    PUT(SYMB_FILE,CHAR);

end if;

-- Get the next character in the file.
GET(SDF_FILE,CHAR);

end loop;

-- Processing of the SDF file is finished. Both of the files may now
-- be closed.
CLOSE( SDF_FILE );
CLOSE( SYMB_FILE );

end CONVERT_SDF_TO_SYMB;

.
.
.
```

Using the FORM Parameter

The UCS Runtime System raises the exception `USE_ERROR` if the `FORM` string is not correct or if a non-supported attribute for a given package is used.

An example of the `FORM` parameter (for `DIRECT_IO`) follows:

```
FORM=>  "Block_size => 3584, Record_size => 448, Append => True, " &
        "Max_Rec_Num => 500, Skeletonize => True, Shared => True, " &
        "Prep_Factor => 448, Labels => Omitted, Disposition => Rewind"
```

The `FORM` parameter must use named notation for each attribute. The qualifier is to the left of the `'=>'`. If the qualifier is omitted from the `FORM` string, the UCS Runtime System raises the exception `USE_ERROR`.

Access Protection

The OS 1100 Executive (Exec) provides access protection of external files by means of access control records (ACR). The system uses these records to define the privileges that the owner of the file has granted to other users.

Files created by Ada programs have default protection values. It is the responsibility of the user to see that any files that have been created with no access protection keys are copied into files that have the level of protection desired, or that the system administrator changes the attributes of the file created.

Another option is to use the system processor called the Site Management Complex (SIMAN). This tool can redefine the security levels associated with files that are created by executing UCS Ada programs.

Closing a File Explicitly

The UCS Runtime System uses the Processor Common Input/Output System (PCIOS) to produce data files that are compatible between high-level processors, utilities, and query systems. Programs access these common I/O modules through a UCS Runtime System interface.

Additionally, a the UCS Runtime System epilogue performs a function that ensures that all files (except `STANDARD_INPUT` and `STANDARD_OUTPUT`) are closed at program termination. The Ada standard does not define what happens when a program terminates without closing all the opened files. Thus, a program that depends on this support feature of the UCS Runtime System may have problems when ported to another system.

If the execution of a program encounters a hard I/O device error, an exception is raised and the UCS Runtime System epilogue attempts to close any files that are open. However, all open files will remain open at program termination when the I/O device experiences a hard error.

Limitations on Procedure RESET

An internal file opened for input can be reset for output; however, the contents of the file may be deleted.

F.8.2. Input/Output Packages

The following table describes the implementation dependencies of the UCS Ada input/output packages.

Package	Dependency
SEQUENTIAL_IO (Generic)	Is target-independent, except for the FORM parameter. There are no implementation dependencies within the generic package SEQUENTIAL_IO.
DIRECT_IO (Generic)	Is target-independent, except for the FORM parameter. The following defines all of the implementation-dependent items associated with the generic package DIRECT_IO: type COUNT is INTEGER range 0 .. 2**18 - 1;
TEXT_IO	Is target independent, except for the FORM parameter. The following defines all of the implementation-dependent items associated with the TEXT_IO package: type COUNT is INTEGER range 0 .. 2**18 - 1; subtype FIELD is INTEGER range 0 .. 132;
LOW_LEVEL_IO	The Ada standard outlines the package LOW_LEVEL_IO, which is concerned with low-level machine-dependent input/output. This type of input/output can be used, for example, to write to device drivers or access device registers. The package LOW_LEVEL_IO is not implemented in UCS Ada. You can, however, use interfaces to subprograms written in the other UCS languages as an alternative method.

F.8.3. IO_EXCEPTIONS

UCS Ada implements IO_EXCEPTIONS as defined the Ada standard (see Section 14.5 of the *UCS Ada Programming Reference Manual Volume 1*). This package defines the exceptions needed by the packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

F.8.4. I/O and Tasking

It is recommended that all of a program's I/O be done in one task. It is possible to do I/O in more than one task as long as each task does I/O to a separate (it's own) file. If a program does not follow this convention, we cannot guarantee correct results or correct order in the file for the program.

F.9. Characteristics of Data Types

The following subsections describe the implementation dependencies of UCS Ada data types.

F.9.1. Integer Types and Attributes

The ranges of values for integer types declared in package STANDARD are as follows:

```
type INTEGER is range -34_359_738_367 .. 34_359_738_367;  -- 2**35-1
```

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

```
type COUNT is range 0 .. 262_143;  -- 2**18-1
type POSITIVE_COUNT is range 1 .. COUNT'LAST;  -- 2**18-1
```

For the package TEXT_IO, the range of values for the type FIELD is as follows:

```
subtype FIELD is INTEGER range 0 .. 132;
```

F.9.2. Floating-Point Types and Attributes

Type FLOAT

Attribute	Exact Value	Approximate Value
DIGITS	7	
MANTISSA	25	
EMAX	100	
EPSILON	2.0^{-24}	5.96E-08
SMALL	2.0^{-101}	3.94E-31
LARGE	$2.0^{100} * (1.0 - 2.0^{-25})$	1.27E+30
SAFE_EMAX	127	
SAFE_SMALL	2.0^{-128}	2.94E-39
SAFE_LARGE	$2.0^{127} * (1.0 - 2.0^{-25})$	1.70E+38
FIRST	$-2.0^{127} * (1.0 - 2.0^{-27})$	-1.70E+38
LAST	$2.0^{127} * (1.0 - 2.0^{-27})$	1.70E+38
MACHINE_RADIX	2	
MACHINE_MANTISSA	27	
MACHINE_EMAX	127	
MACHINE_EMIN	-128	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOW	TRUE	
SIZE	36	

Implementation-Dependent Language Features

Type LONG_FLOAT

Attribute	Exact Value	Approximate Value
DIGITS	17	
MANTISSA	58	
EMAX	232	
EPSILON	2.0^{-57}	6.94E-18
SMALL	2.0^{-233}	7.24E-71
LARGE	$2.0^{232} * (1.0-2.0^{-58})$	6.90E+69
SAFE_EMAX	1023	
SAFE_SMALL	2.0^{-1024}	5.56E-309
SAFE_LARGE	$2.0^{1023} * (1.0-2.0^{-58})$	8.99E+307
FIRST	$-2.0^{1023} * (1.0-2.0^{-60})$	-8.99E+307
LAST	$2.0^{1023} * (1.0-2.0^{-60})$	8.99EE+307
MACHINE_RADIX	2	
MACHINE_MANTISSA	60	
MACHINE_EMAX	1023	
MACHINE_EMIN	-1024	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	72	

F.9.3. Attributes of Type DURATION

type DURATION is delta 2.0^{-18} range -86_400.0 .. 86_400.0;

Attribute	Exact Value
DURATION'DELTA	2.0^{-18}
DURATION'SMALL	2.0^{-18}
DURATION'SAFE_LARGE	131071.9999
DURATION'FIRST	-86400.0
DURATION'LAST	86400.0

F.9.4. Default Sizes and Alignments

The following table shows the default representation of non-composite data types in UCS Ada.

Type	Default Size (bits)	Default Alignment	Size clauses allowed?
Enumeration	36	word	Yes. Must be between logical size and 36
Character	9	word	Yes. Must be between logical size and 36
Boolean	9	word	Yes. Must be between logical size and 36
Integer	36	word	Yes. Must be between logical size and 36
Float	36	word	Must be the default size
Long_Float	72	word	Must be the default size
Fixed point	36	word	Yes. Same as integer
Access	36	word	Must be 36
Task	36	word	Must be 36

F.10. Other Implementation-Dependent Characteristics

The following table describes other miscellaneous implementation characteristics.

Characteristic	Description
Heap	<p>Objects that have allocated space from the heap have the following common characteristics:</p> <ul style="list-style-type: none"> • All are part of collections that contain objects that are either fixed or variable in their size • All are associated with a scope. <p>The scope can be as broad as global or as limited as a specific routine. All Ada data that is allocated via an allocator (NEW) is allocated on the heap. In addition, the compiler may decide to allocate other data on the heap.</p> <p>The source of storage for objects in a program is dependent upon the value of the compile-time constants STACK and DYNAMIC. The STACK and DYNAMIC keyword options in UADA (see Table -) are available to modify the constraints on the static and dynamic stacks. By controlling the size of the objects that are put on these stacks, you can indirectly control which objects are put onto the heap.</p> <p>The ADABND processor has a set of commands that are available to control the definition of the heap's initial size, maximum size, and increment size during execution. These are described in .</p>
Task	<p>In UCS Ada level 1R1, there is a one-to-one correspondence of Ada tasks to OS 1100 activities. See for a list of the ADABND processor commands available to control task characteristics.</p>
Exception	<p>Because of the current interpretation of the Ada standard, code generated by UCS Ada does not produce NUMERIC_ERROR exceptions. Instead, numeric error cases such as overflow and divide by zero produce CONSTRAINT_ERROR exceptions.</p> <p>A NUMERIC_ERROR exception can only be generated by using the following Ada statement:</p> <pre>RAISE NUMERIC_ERROR;</pre>

continued

continued

Characteristic	Description
Input/Output	All files created by an executing Ada program are compatible with the other UCS languages. The different UCS languages can share I/O files; however, each language that calls a file must close it before another UCS language can access it. See Section for more information.
Main Subprogram	A library unit can be used as a main subprogram if and only if it is a procedure that is not generic and has no formal parameters.
Compilation Units	UCS Ada supports the instantiation of a generic compilation unit whose body has not yet been compiled (see compiler option NO-GEN-INL in Table -).

Table 6-1. ADAREFORMAT Keyword Options

Keyword	Description
KEY/LOWER (default) KEY/UPPER	KEY/LOWER converts all Ada keywords to lowercase. KEY/UPPER converts all Ada keywords to uppercase. If you do not specify this keyword, the default is KEY/LOWER.
IDENT/CAP IDENT/KEEP IDENT/LOWER IDENT/UPPER (default)	IDENT/CAP capitalizes the first letter of each word in an identifier that uses underscores (For example, Get_Name). All other letters in the identifier are lowercase. IDENT/KEEP keeps ADAREFORMAT from converting the identifiers. All of the identifiers in the source remain the same. IDENT/LOWER converts all of the identifiers in the Ada source to lowercase. IDENT/UPPER converts all of the identifiers in the Ada source to uppercase. If you do not specify this keyword, the default is IDENT/UPPER.
TAB/ <i>n</i>	ADAREFORMAT uses the value of <i>n</i> as the number of space characters to indent the reformatted source. The value of <i>n</i> must be a positive integer in the range $2 \leq n \leq 6$. If you do not specify a value, the default value of <i>n</i> is 3 spaces.

6.1.2. Using Pragma INDENT

This pragma effects the reformatting of the Ada source by the ADAREFORMAT processor. It has no effect, however, on the compilation of a UCS Ada unit.

The format and definition of pragma INDENT are as follows:

```
pragma INDENT(ON)    -- Allows ADAREFORMAT to format the source
pragma INDENT(OFF)   -- Prevents ADAREFORMAT from formatting the source
```

The OFF format inhibits any manipulation of the Ada source by the ADAREFORMAT processor preventing ADAREFORMAT from formatting all source following the pragma. This continues until a pragma INDENT(ON) is encountered in the source, which causes ADAREFORMAT to resume its normal reformatting actions in the source after the pragma.

6.1.3. Sample of Reformatting Sou

To be supplied.

← Pragma INDENT
(also mentioned in
Appendix F)

1 of 54 of
 referenced on pg. 754 of
 Appendix 4
 ↓

Keyword Options for Program Control (cont.)

Description	
<p>does not allow inline expansion of functions and subprogram calls.</p> <p>by SOURCE or the S letter option, the listing generated lines that are expanded.</p> <p>n specifies the machine class on which the object module produced will execute. This overrides the default that is defined at the time the Support System (LSS) is installed.</p> <p>one of the following values, depending on the machine:</p> <ul style="list-style-type: none"> extended mode machine 90 and 2200/600 system 200, 2200/300, and 2200/400 system <p>specifies that the object module produced by the compiler executes in the class defined at the time LSS was installed.</p>	
GEN-INL (default) NO-GEN-INL	<p>GEN-INL causes the compiler to place the code of a generic instantiation within the compilation unit that contains the instantiation (inline).</p> <p>NO-GEN-INL causes the compiler to place the code of a generic instantiation in a separate subunit. Use this option to manage the size of large compilation units.</p>
GEN-INST NO-GEN-INST (default)	<p>GEN-INST causes the compiler to instantiate a pending uncompiled generic body whose template has been compiled or recompiled in another library (specified by the library field). See 5.1.2 for a discussion of pending instantiation units.</p> <p>Initiate the instantiation with the following on the @UADA compiler call:</p> <ul style="list-style-type: none"> Specify the GEN-INST keyword option on the @UADA processor call Omit the source-program field and the l-option (there is no source to process) <p>For example,</p> <pre>@UADA .qual'library....GEN-INST</pre>

Referenced
in App. F.

Section 9 Interlanguage Calls

Procedures written in other UCS languages (currently UCS C, UCS COBOL, UCS FORTRAN, and UCS Pascal) can be called from UCS Ada, as follows:

- All communication must occur by means of parameters and function results.
- You must include a pragma INTERFACE for each subprogram written in another language.
- Calls from Ada to other UCS languages use the standard calling sequence (SCS).
- Other UCS languages cannot call Ada code. Therefore, an Ada application must begin execution in an Ada main program.
- Other UCS languages cannot access Ada data, except through parameters passed from Ada code. That is, there is no way to export Ada static global data.

However, differences among the languages for handling the following items require certain restrictions on passing arguments:

- Data types
- Data alignments
- Data sizes
- Passing modes used by the languages

- Pass by value (PBV)

The convention of passing a parameter value directly as an immediate value

In the UCS standard calling sequence (SCS), there are cases where the value can be immediate if its size is two words or less. If the value to be passed is greater than two words, the SCS always uses the pass-by-reference-value format.

- Pass by reference (PBR)

The convention of passing the address of an object

This allows the called program to change the parameter, if the language rules allow it.

Interlanguage Calls

- **Pass by reference value (PBRV)**

The convention where the compiler automatically copies an object to a temporary variable and passes the address of the temporary variable

The called program cannot change the value of the object, only the value of the copy. This method is used by SCS when the value of the parameter to be passed is greater than two words in size (that is, it does not fit into the SCS packet as an immediate value).

- **Pass by copy (PBC)**

The convention where the compiler automatically copies an object to a temporary variable, passes the address of the temporary variable, and copies the value of the temporary to the object on return

This allows the called program to change the parameter. Ada requires this type of parameter passing for OUT and IN OUT scalars.

These items are discussed in subsections later in the section, which show UCS Ada calling a particular language.

9.1. Using Pragma INTERFACE

You must include a pragma INTERFACE for each subprogram that is written in another language and called from your Ada program. Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which parameter passing conventions must be generated.

Format

```
pragma INTERFACE(language_name, subprogram_name);
```

where:

language_name

is one of the UCS languages (UC, UCOB, UFTN, UPAS). These UCS languages use the standard calling sequence (SCS) for external interfaces.

subprogram_name

is the name by which the Ada program references the external subprogram.

If *subprogram_name* differs from the external name of the interfaced subprogram, you can associate *subprogram_name* with the external name by specifying pragma INTERFACE_NAME (see 9.1.1) in conjunction with pragma INTERFACE.

When using pragma INTERFACE, case sensitivity of the different UCS languages makes the following items important:

- The compiler internally converts all lowercase letters in identifiers (such as *subprogram_name*) to uppercase
- External names generated by UCS C are case sensitive, while those generated by other UCS languages are not

Therefore, you must use pragma INTERFACE_NAME if the external name of a subprogram generated by UCS C contains lowercase characters. See 9.1.2 for examples of using pragma INTERFACE as well as pragma INTERFACE_NAME.

9.1.1. Using Pragma INTERFACE_NAME

Use this pragma with pragma INTERFACE when the Ada name of the subprogram that the program calls is not identical to the external subprogram name. This pragma associates the two names.

Format

```
pragma INTERFACE_NAME(ada_subprogram_name, external_language_name);
```

where:

ada_subprogram_name

specifies the name of the subprogram as defined in the Ada program.

Interlanguage Calls

external_language_name

is a string literal (surrounded by quotes) that specifies the name of the subprogram as defined in the external program. The specified name of a UCS C routine must identically match the external C name (this includes the use of lowercase letters). External names for all of the other UCS languages must be uppercase.

You can use pragma `INTERFACE_NAME` to identify routines in other languages that are not named with legal Ada identifiers. Ada identifiers can only contain letters, digits, or underscores, whereas the Linking System allows external names to contain other characters such as the dollar sign (\$). These characters can be specified as the *external_language_name* argument of the Ada subprogram.

Pragma `INTERFACE_NAME` is allowed at the same places of an Ada program as pragma `INTERFACE` (see the *UCS Ada Programming Reference Manual Volume 1*). However, pragma `INTERFACE_NAME` must always occur immediately after the pragma `INTERFACE` declaration of the interfaced subprogram.

9.1.2. Examples

Example 1

The following code shows the declaration of a UCS COBOL subprogram, externally named `SAMPLE`, that an Ada program calls. The name of the external subprogram is also `SAMPLE`, so pragma `INTERFACE_NAME` is not used in the declaration.

```
procedure SAMPLE(I : INTEGER);  
pragma INTERFACE(UCOB,SAMPLE);
```

Example 2

The following code shows an interfaced subprogram specification. The addition of pragma `INTERFACE_NAME` declares that the UCS FORTRAN routine with external name `GETNAM` is referenced in the UCS Ada program as procedure `GET_NAME`.

```
procedure GET_NAME(I : INTEGER);  
pragma INTERFACE(UFTN,GET_NAME);  
pragma INTERFACE_NAME(GET_NAME,"GETNAM");
```

Example 3

The following code shows a subprogram specification for a UCS C subprogram. The addition of pragma `INTERFACE_NAME` declares that the UCS C routine with external name `test` is referenced in the UCS Ada program as procedure `TEST`.

```
procedure TEST(I : INTEGER);  
pragma INTERFACE(UC,TEST);  
pragma INTERFACE_NAME(TEST,"test");
```

Note: If pragma `INTERFACE_NAME` is not used when calling UCS C subprograms, the imported name of the UCS C subprogram uses all uppercase characters.

9.2. Restrictions on Calling UCS Languages

The following subsections describe the interfaces to subprograms written in other UCS languages. Each language has specific interface rules. However, the following list contains general rules for calling other languages from Ada code:

- You must be aware of storage sizes and alignment differences in the called language. In some cases, you must use a representation clause to properly size and align the data to the exact format required. This is especially true for Ada composite types (array and record) and data referenced by Ada access types. You may have to examine the allocation listings produced by the UCS languages to determine the exact layout specifications.
- Alignment of parameter and function result data is as follows:
 - Scalars, including the predefined type character, are right-justified.
 - Composites, such as arrays, strings, and records, are left-justified.
- If the actual parameter is not aligned on a word boundary, the compiler generates code to move it to a temporary variable. In addition, if the size of the actual parameter does not match the size of the Ada formal parameter, the actual parameter is moved to a temporary variable.
- You must ensure that an Ada scalar formal parameter whose size is not a multiple of 36 bits matches the alignment of the formal parameter in the other language. For example, if the Ada scalar formal parameter is a 9-bit integer type, you must ensure it is passed in a way that is expected by the called language.

One way to avoid problems is to declare an interface scalar formal parameter in both Ada and the called language as a multiple of 36 bits.

- A character data item is considered to be a scalar (similar to an integer) whose value is right-justified.
- Function result types that are Ada arrays, including strings, and records must be constrained.
- Parameter types need not be constrained. However, for unconstrained parameter types, no descriptor information is passed to the calling program that describes the parameter, as is done for noninterface calls.
- When passing Ada records, you must account for the internal format of discriminants and compiler-generated record components. In addition, you may have to use pragma IMPROVE (see F.1.2) in some cases to eliminate internal fields in records. You can use an Ada representation clause to designate record formats.
- The index types of Ada arrays must be mapped properly onto the index types of the called program. For example, FORTRAN array indexes are integers. Arrays are always PBR or PBRV.

Note: FORTRAN arrays are stored in column-major order, while Ada stores arrays in row-major order.

- The values for enumeration literals in Ada enumeration types must be mapped properly onto the values expected by the called program. You can use an Ada representation clause to designate enumeration values.
- The formal parameter type must match the actual parameter type. For example, if you pass an Ada enumeration variable to a FORTRAN INTEGER variable (since FORTRAN has no enumeration type), the Ada formal and actual parameter types must be enumeration.
- Ada access variables point only to word-aligned objects. That is, there is no bit offset in the pointer.

To get an 1100 extended mode virtual address (VA) in an Ada access variable, include the following specification on the access type (otherwise, the access variable contains a heap bank offset):

```
PRAGMA NON_ADA_ACCESS
```

Use the pragma in cases where an access variable parameter is passed to another language that expects a VA.

Note: A heap bank offset has no meaning for any other high-level language called.

Another way to pass a virtual address (VA) on an interface call, other than with pragma INTERFACE (UC), is the following:

- Declare the Ada formal parameter as an integer.
- Pass *object*'ADDRESS as the actual parameter, where *object* is the variable whose address is passed.
- You can pass an object with an Ada private type; the actual type of the object is used.
- You cannot pass Ada fixed-point or task types on any pragma INTERFACE call.
- You cannot pass subprogram names as parameters.
- You should not pass null records, arrays, or strings as parameters.
- In many cases, the compiler copies parameter data to a word-aligned temporary variable. When necessary, it also copies the temporary variable back to the original variable.
- If the non-Ada subprogram is a function, its value is returned to Ada by standard calling sequence (SCS) conventions. The compiler then implicitly converts the returned value to the Ada calling sequence (ACS) format.
- Ada does not allow a function to have OUT or IN OUT parameters. However, this is implicitly allowed in other languages.
 - One way to handle this problem is to have the Ada program call a procedure in the other language, which then calls the desired function in that language. The procedure has an extra OUT parameter to contain the function result.

Example

Ada declarations:

```
X1: INTEGER;  
X2: INTEGER;  
procedure P(M: out INTEGER;  
            I: out INTEGER)  
pragma INTERFACE(UFTN,P);
```

Ada call:

```
P(X1,X2);  
-- X1 is the FORTRAN function result.  
-- X2 is the OUT parameter whose value was changed in the function.
```

FORTRAN subroutine, which calls the function:

```
subroutine P(M,I)  
M = N(I)  
return  
end
```

FORTRAN function:

```
function N(J)  
J = ...  
N = ...  
return  
end
```

- Another way to handle the problem in certain interface languages is to pass a pointer as an IN parameter to the function, then use the pointer in the called program to change the data.
- Ada does not use the compile-time service routine to pass parameters, since this feature pertains only to certain UCS languages.
- An I/O file can be shared between different UCS languages, but it must be closed by the language that opened it before it can be accessed by the other language.

9.3. Calling Other UCS Languages from UCS Ada

Subsections 9.3.1 through 9.6 each contain a table that lists Ada data types and the corresponding types of the other language, if any. Codes in the table identify the possible relations between corresponding types, as follows:

- Y

The type is allowed with no restrictions.

- N

The type is not allowed.

- YL

The type is allowed, but the data item must have the proper storage layout (alignment and size) required by the called program, including alignment and exact data size for Ada composites. You can use an Ada representation clause, if necessary, for an Ada record.

- YT

The argument can be passed, but one of the following is true:

- Ada does not have a type that directly corresponds to the type of the called language.
- The called language does not have a type that directly corresponds to the Ada type.

You are responsible for setting up the proper layout and values expected by the called program. For example,

- The COMPLEX type in FORTRAN can be represented in Ada as a record.
- The enumeration type in Ada can be represented in FORTRAN as an integer.

You can use an Ada representation clause, if necessary.

- YV

The type is allowed, but the value passed (such as the value for an enumeration type) must be in the range of values required by the called program. You can use an Ada representation clause, if necessary.

- YP

The pointer (one word in length) can be passed. However, if an Ada access value is passed to a pointer in another language, you must include the following specification on the access type:

```
PRAGMA NON_ADA_ACCESS
```

This allows you to get the virtual address (VA) format of the pointer.

Interlanguage Calls

Another way to pass a virtual address (VA) on an interface call is the following:

- Declare the Ada formal parameter as an integer.
- Pass *object*'ADDRESS as the actual parameter, where *object* is the variable whose address is passed.

The format parameter in the called language can then be a pointer type.

- **WD**

The type is allowed, but size of the object must be word (36 bits). You can use an Ada representation clause, if necessary.

- **2WD**

The type is allowed, but size of the object must be double word (72 bits). You can use an Ada representation clause, if necessary.

- **4WD**

The type is allowed, but size of the object must be quadruple word (144 bits). You can use an Ada representation clause, if necessary.

Any special considerations of which you need to be aware when calling another UCS language from UCS Ada are listed after the data type table for each language. These considerations are in addition to the general set of restrictions listed in 9.2.

9.3.1. UCS Ada Calling UCS C

Table 9-1 lists the data types defined by Ada and indicates the corresponding C data type, if any.

Table 9-1. Ada vs C Data Types and Attributes

Ada Actual Argument	C Parameter	Relation
integer	int signed int long long int signed long signed long int	YL ***
float	float	WD
long_float	double long double	2WD
access	pointer	WD *
enumeration	int	YL, YV ***
boolean	int	YL, YT
character	int	YL ***
string	string	Y *, **
array	array	YL *
record	struct	YL

Legend

Y Argument type allowed with no restrictions.

N Argument type not allowed.

WD Size is word.

2WD Size is double word.

YL Argument type allowed, but data item must have proper storage layout.

YT Argument can be passed, but type may not exist in one of the languages and must be handled within called program.

YV Argument type allowed, but value passed must be within range required by called program.

* Ada implicitly converts the Ada pointer to a two-word C pointer. Arrays, including strings, are passed as C pointers, as are OUT and IN OUT parameters. Note that UCS Ada does not convert an expression of the form *object*'ADDRESS to a C pointer.

** You must insert the terminating null character into the Ada code.

*** For a C function in traditional format, you can use types less than one word in length (see Table 9-2).

You should be aware of the following considerations when calling a C program from an Ada program:

- Ada programs can pass parameters to a C function by value or reference, as follows:
 - IN parameters that are scalars (including char objects, which are treated as integers) and small records (less than or equal to two words) are always passed to UCS C by value.
 - OUT and IN OUT parameters that are scalars and small records (less than or equal to two words) are always passed to UCS C by reference. The parameter on the UCS C side should be a pointer (using the * syntax) in this case.
 - Large records (greater than two words) passed to C structures are always passed by reference. The corresponding C parameter is declared as a structure (struct).
 - Arrays (including strings) are passed by reference.
 - Any reference parameter is passed via a C pointer.
- Since the main program is not a C program, the standard C files stdin, stdout, and stderr will not be open. See the UCS C Programming Reference Manual Volume 1 for information on opening the standard C files.
- The called C subprogram cannot have a variable parameter list (a variable number of parameters).
- A pointer in C has a special two-word format. The compiler implicitly converts an access value passed to C to that two-word format.

Note: Only a C pointer to data is supported in an interface call from Ada to C. A C pointer to a subprogram, which is contained in an eight-word structure, is not supported.

- The C convention for indicating the end of a character string (which C maintains as a single-dimension array) is to use a null character, which is a byte containing all zeros. C string functions, as well as the %s format descriptor, require the NULL character to terminate the string.

If an Ada program passes a character string to a C function, it must insert a null character in the appropriate place in the string. If an Ada program receives a character string from a C program, the Ada program should consider the length of the string to be the length of the text preceding the null character in the string, not the declared length of the string.

Examples 9-1 and 9-2 demonstrate how Ada can handle C strings.

- If parameter data is modified in the called C subprogram, the Ada program should pass either of the following:
 - Access value pointing to the data

- **OJT or IN OUT parameters**
- Casting methods (casting a pointer to an integer or an integer to a pointer) for passing parameters to UCS C are not required.
- The called C function should use the prototype format. However, note the following considerations about the format used:
 - Traditional format treats integral data types that are not one word in length as type `int` (that is, as one-word, right-justified entities), as follows:

Table 9-2. Two-Word Data Types That Are Treated as One-Word

C Data Type	Size
short short int signed short signed short int	18 bits
enum	18 bits, unless the values do not fit; otherwise 36 bits
char unsigned char	9 bits

Ada can pass one-word items, (such as integer, enumeration, and character) to these C types in traditional format.

For functions in prototype format, use `int` instead of any of these types to declare formal parameters.

- Traditional format treats type `float` like type `double` (that is, like a two-word item). In prototype format, type `float` is one word in length.
- Traditional format does not affect the processing of nonscalar parameters, such as records (that is, `struct` in C).
- Do not mix prototype and traditional formats in a C function.

Example

This example demonstrates a call from an Ada main program to a C subprogram, with the following actions by the main program and subprogram:

- The Ada main program initializes an integer scalar, string, and a two-dimensional integer array. Each element of the array is initialized with its ordinal value in column-major order.
- The contents of the Ada data items are displayed to verify initialization.
- The Ada main program calls the C subprogram, passing the three data items as actual arguments.

UCS Ada Calling UCS C

- The C subprogram displays the incoming parameters to demonstrate that the parameters were received correctly and that the subprogram interprets them correctly.
- The subprogram updates the contents of the dummy arguments and returns control to the main program. The changes to the string and scalar integer arguments are straightforward. The array is updated by adding the value of the ordinal position of each element to the current contents of each element. Therefore, when the contents of the array are again displayed in the Ada main program, the value of each element should be double the original value.
- The main program again displays the contents of the arguments to verify that the C subprogram updated them correctly.

Figure 9-1 shows the output from executing this program.

```
-- This Ada main program calls a C subprogram.

with TEXT_IO;

procedure CALL_C is

  I, J : INTEGER;

  subtype STR_TYPE is STRING (1 .. 12);
  C : STR_TYPE;

  type ARR_TYPE is array (1 .. 4, 1 .. 3) of INTEGER;
  A : ARR_TYPE;

  procedure CSUB (I1 : in out INTEGER;
                  C1 : in out STR_TYPE;
                  A1 : in out ARR_TYPE);
  pragma INTERFACE (UC, CSUB);

  package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

begin
  TEXT_IO.PUT_LINE (** In Ada main program **);

  -- Intialize and print variables

  I := 123;
  TEXT_IO.PUT (" I = ");
  INT_IO.PUT (I);
  TEXT_IO.NEW_LINE;

  C := "** Ada *";
  TEXT_IO.PUT (" C = ");
  TEXT_IO.PUT_LINE (C);

  TEXT_IO.PUT_LINE (" A = ");
  J := 1;
  for I1 in 1 .. 4 loop
    for I2 in 1 .. 3 loop
      A (I1, I2) := J;
      INT_IO.PUT (A (I1, I2));
      J := J + 1;
    end loop;
    TEXT_IO.NEW_LINE;
  end loop;

  -- Call UC
```

UCS Ada Calling UCS C

```
CSUB (I, C, A);

-- Print variables on return from UC

TEXT_IO.NEW_LINE;
TEXT_IO.PUT_LINE ("** Back in Ada main program **");

TEXT_IO.PUT (" I = ");
INT_IO.PUT (I);
TEXT_IO.NEW_LINE;

TEXT_IO.PUT (" C = ");
TEXT_IO.PUT_LINE (C);

TEXT_IO.PUT_LINE (" A = ");
J := 1;
for I1 in 1 .. 4 loop
  for I2 in 1 .. 3 loop
    INT_IO.PUT (A (I1, I2));
  end loop;
  TEXT_IO.NEW_LINE;
end loop;

end CALL_C;
```

Example 9-1. UCS Ada Main Program to Call UCS C Subprogram

```
#include <stdio.h>
#include <string.h>

void CSUB(int *iparm, char *cparm, int (*aparm)[4][3])
{
    FILE *outfile ;

    int i, j ;

    outfile = fopen("/dev/tty","w") ;
    fprintf(outfile," \n") ;
    fprintf(outfile,"* In C subprogram * \n") ;
    fprintf(outfile," iparm = %d\n", *iparm) ;
    fprintf(outfile," cparm = %s\n", cparm) ;
    fprintf(outfile," aparm = \n") ;

    for (i=0; i<4; i++)
    {
        fprintf(outfile," ");
        for (j=0; j<3; j++)
            fprintf(outfile,"%d ",(*aparm)[i][j]) ;
        fprintf(outfile,"\n") ;
    }
    fclose(outfile) ;

    *iparm = 456 ;
    strcpy(cparm,"* C * ") ;

    for (i=0; i<4; i++)
        for (j=0; j<3; j++)
            (*aparm)[i][j] = (*aparm)[i][j] + (j+1) + (i*3) ;
}
```

Example 9-2. UCS C Subprogram Called from UCS Ada Main Program

UCS Ada Calling UCS C

```
* In Ada main program *
I =      123
C = * Ada *
A =
      1      2      3
      4      5      6
      7      8      9
     10     11     12

* In C subprogram *
iparm = 123
cparm = * Ada *
aparm =
      1  2  3
      4  5  6
      7  8  9
     10 11 12

* Back in Ada main program *
I =      456
C = * C *
A =
      2      4      6
      8     10     12
     14     16     18
     20     22     24
```

Figure 9-1. Output of UCS Ada Program Calling UCS C Subprogram

9.4. UCS Ada Calling UCS COBOL

Table 9-3 lists the data types defined by Ada and indicates the corresponding COBOL data type, if any.

Table 9-3. Ada vs COBOL Data Types and Attributes

Ada Actual Argument	COBOL Argument	Relation
integer	binary	YL
float	comp-1 computational-1	WD
long_float	comp-2 computational-2	2WD
access	(none)	N
enumeration	binary	YL, YT
boolean	binary	YL, YT
character	binary	YL
string	disp display	Y
array	occurs	YL
record	record	YL

Legend

Y Argument type allowed with no restrictions.

N Argument type not allowed.

WD Size is word.

2WD Size is double word.

YL Argument type allowed, but data item must have proper storage layout.

YT Argument can be passed, but type may not exist in one of the languages and must be handled within called program.

You should be aware of the following considerations when calling a COBOL program from an Ada program:

- Ada passes parameters to COBOL by reference (PBR).
- Ada output to the standard print file operates in the reverse order of the COBOL DISPLAY statement. Ada performs a line feed and then writes the items in the output list. By default, the COBOL DISPLAY statement first writes the items in the output list and then performs a line feed. Therefore, if an Ada program outputs a line and then calls a COBOL program that uses a DISPLAY statement to output the next line, the COBOL program must output an initial blank line to avoid overwriting the Ada line. The example at the end of this subsection demonstrates this consideration.
- COBOL identifiers are limited to a length of 30 characters. They cannot contain an underscore (_) or a dollar sign (\$).
- A COBOL subprogram cannot be invoked as a function because the concept of a function does not exist in the COBOL language.

Example

This example demonstrates a call from an Ada main program to a COBOL subprogram, with the following actions by the main program and subprogram:

- The Ada main program initializes an integer scalar, a string, and a two-dimensional integer array. Each element of the array is initialized with its ordinal value in column-major order.
- The contents of the Ada data items are displayed to verify initialization.
- The Ada main program calls the COBOL subprogram, passing the three data items as actual arguments.
- The COBOL subprogram displays the incoming dummy arguments to demonstrate that the dummy arguments were received correctly and that the subprogram interprets them correctly.
- The subprogram updates the contents of the dummy arguments and returns control to the main program. The changes to the string and integer scalar arguments are straightforward. The array is updated by adding the value of the ordinal position of each element to the current contents of each element. Therefore, when the contents of the array are displayed again in the Ada main program, the value of each element should be double the original value.
- The main program again displays the contents of the arguments to verify that the COBOL subprogram updated them correctly.

Figure 9-2 shows the output from executing this program.

```
-- This Ada main program calls a COBOL subprogram.

with TEXT_IO;

procedure CALLCOB is

    I, J : INTEGER;

    subtype STR_TYPE is STRING (1 .. 12);
    C : STR_TYPE;

    type ARR_TYPE is array (1 .. 4, 1 .. 3) of INTEGER;
    A : ARR_TYPE;

    procedure COBSUB (I1 : in out INTEGER;
                     C1 : in out STR_TYPE;
                     A1 : in out ARR_TYPE);
    pragma INTERFACE (UCOB, COBSUB);

    package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

begin
    TEXT_IO.PUT_LINE ("** In Ada main program **");

    -- Initialize and print variables

    I := 123;
    TEXT_IO.PUT (" I = ");
    INT_IO.PUT (I);
    TEXT_IO.NEW_LINE;

    C := "** Ada * ";
    TEXT_IO.PUT (" C = ");
    TEXT_IO.PUT_LINE (C);

    TEXT_IO.PUT_LINE (" A = ");
    J := 1;
    for I1 in 1 .. 4 loop
        for I2 in 1 .. 3 loop
            A (I1, I2) := J;
            INT_IO.PUT (A (I1, I2));
            J := J + 1;
        end loop;
        TEXT_IO.NEW_LINE;
    end loop;

    -- Call UCOB
```

UCS Ada Calling UCS COBOL

```
COBSUB (I, C, A);

-- Print variables on return from UCOS

TEXT_IO.NEW_LINE;
TEXT_IO.PUT_LINE (** Back in Ada main program **);

TEXT_IO.PUT (" I = ");
INT_IO.PUT (I);
TEXT_IO.NEW_LINE;

TEXT_IO.PUT (" C = ");
TEXT_IO.PUT_LINE (C);

TEXT_IO.PUT_LINE (" A = ");
J := 1;
for I1 in 1 .. 4 loop
  for I2 in 1 .. 3 loop
    INT_IO.PUT (A (I1, I2));
  end loop;
  TEXT_IO.NEW_LINE;
end loop;

end CALLCOB;
```

Example 9-3. UCS Ada Main Program to Call UCS COBOL Subprogram

```

IDENTIFICATION DIVISION.
PROGRAM-ID.
    COBSUB.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. UNISYS-2200.
OBJECT-COMPUTER. UNISYS-2200.
SPECIAL-NAMES.
    PRINTER IS PRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 i PIC 9(10) BINARY.
01 j PIC 9(10) BINARY.
LINKAGE SECTION.
01 parm1 PIC S9(10) USAGE BINARY.
01 parm2 PIC X(12) USAGE DISPLAY.
01 workrec3.
    02 row OCCURS 4 TIMES.
        03 col PIC S9(10) USAGE BINARY OCCURS 3 TIMES.
PROCEDURE DIVISION USING parm1, parm2, workrec3.
BEGIN.
    DISPLAY ' '.
    DISPLAY ' '.
    DISPLAY '* In COBOL subprogram *'.
    DISPLAY '   Parm1 = ' parm1.
    DISPLAY '   Parm2 = ' parm2.
    DISPLAY '   Parm3 = '.
    PERFORM VARYING i FROM 1 BY 1 UNTIL i > 4
        DISPLAY '           ' col(i, 1) col(i, 2) col(i, 3)
    END-PERFORM.

    MOVE 456 TO parm1.
    MOVE '* COBOL *' TO parm2.
    PERFORM VARYING i FROM 1 BY 1 UNTIL i > 4
        PERFORM VARYING j FROM 1 BY 1 UNTIL j > 3
            COMPUTE col(i, j) = col(i, j) + j + ((i - 1) * 3)
        END-PERFORM
    END-PERFORM.

```

Example 9-4. UCS COBOL Subprogram Called from UCS Ada Main Program

UCS Ada Calling UCS COBOL

```
* In Ada main program *
I =      123
C = * Ada *
A =
      1      2      3
      4      5      6
      7      8      9
     10     11     12

* In COBOL subprogram *
Parm1 = +0000000123
Parm2 = * Ada *
Parm3 =
+0000000001+0000000002+0000000003
+0000000004+0000000005+0000000006
+0000000007+0000000008+0000000009
+0000000010+0000000011+0000000012

* Back in Ada main program *
I =      456
C = * COBOL *
A =
      2      4      6
      8     10     12
     14     16     18
     20     22     24
```

Figure 9-2. Output of UCS Ada Program Calling UCS COBOL Subprogram

9.5. UCS Ada Calling UCS FORTRAN

Table 9-4 lists the data types defined by Ada and indicates the corresponding FORTRAN data type, if any.

Table 9-4. Ada vs FORTRAN Data Types and Attributes

Ada Actual Argument	FORTRAN Argument	Relation
integer	integer	WD
float	real	WD
long_float	double precision	2WD
access	(none)	N
enumeration	integer	WD, YT
boolean	logical	WD
character	integer	WD
string	character	Y *
array	array	YL
record	(none)	YT

Legend

Y Argument type allowed with no restrictions.

N Argument type not allowed.

WD Size is word.

2WD Size is double word.

YL Argument type allowed, but data item must have proper storage layout.

YT Argument can be passed, but type may not exist in one of the languages and must be handled within called program.

* The character length in the SCS parameter packet (third word) is always zero. Therefore, the FORTRAN formal parameter cannot be declared as CHARACTER(*).

UCS Ada Calling UCS FORTRAN

You should be aware of the following considerations when calling a FORTRAN program from an Ada program:

- Ada passes parameters to UCS FORTRAN by reference (PBR).
- A UCS FORTRAN function called from UCS Ada cannot be of COMPLEX*16.
- UCS FORTRAN arrays are allocated storage in column-major order. UCS Ada allocates arrays in row-major order. If an Ada array is passed as an argument to UCS FORTRAN, the FORTRAN program must declare and reference the array with the array subscripts in reverse order as compared to the array in Ada. In addition, the UCS FORTRAN array formal parameter must be in contiguous storage.

Example

This example demonstrates a call from an Ada main program to a FORTRAN subprogram, with the following actions by the main program and subprogram:

- The Ada main program initializes an integer scalar, a string, and a two-dimensional integer array. Each element of the array is initialized with its ordinal value in column-major order.
- The contents of the Ada data items are displayed to verify initialization.
- The Ada main program calls the FORTRAN subprogram, passing the three data items as actual arguments.
- The FORTRAN subprogram displays the incoming dummy arguments to demonstrate that the dummy arguments were received correctly and that the subprogram interprets them correctly. Note that the order of the array bounds is reversed from the Ada program.
- The subprogram updates the contents of the dummy arguments and returns control to the main program. The changes to the string and scalar integer arguments are straightforward. The array is updated by adding the value of the ordinal position of each element to the current contents of each element. Therefore, when the contents of the array are displayed again in the Ada main program, the value of each element should be double the original value.
- The main program again displays the contents of the arguments to verify that the FORTRAN subprogram updated them correctly.

Figure 9-3 shows the output from executing this program.

```
-- This Ada main program calls a FORTRAN subprogram.

with TEXT_IO;

procedure CALLFOR is

    I, J : INTEGER;

    subtype STR_TYPE is STRING (1 .. 12);
    C : STR_TYPE;

    type ARR_TYPE is array (1 .. 4, 1 .. 3) of INTEGER;
    A : ARR_TYPE;

    procedure FORSUB (I1 : in out INTEGER;
                     C1 : in out STR_TYPE;
                     A1 : in out ARR_TYPE);
    pragma INTERFACE (UFTN, FORSUB);

    package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

begin
    TEXT_IO.PUT_LINE ("* In Ada main program *");

    -- Initialize and print variables

    I := 123;
    TEXT_IO.PUT (" I = ");
    INT_IO.PUT (I);
    TEXT_IO.NEW_LINE;

    C := "* Ada *";
    TEXT_IO.PUT (" C = ");
    TEXT_IO.PUT_LINE (C);

    TEXT_IO.PUT_LINE (" A = ");
    J := 1;
    for I1 in 1 .. 4 loop
        for I2 in 1 .. 3 loop
            A (I1, I2) := J;
            INT_IO.PUT (A (I1, I2));
            J := J + 1;
        end loop;
        TEXT_IO.NEW_LINE;
    end loop;

    -- Call UFTN
```

UCS Ada Calling UCS FORTRAN

```
FORSUB (I, C, A);

-- Print variables on return from UFTN

TEXT_IO.NEW_LINE;
TEXT_IO.PUT_LINE ("** Back in Ada main program **");

TEXT_IO.PUT (" I = ");
INT_IO.PUT (I);
TEXT_IO.NEW_LINE;

TEXT_IO.PUT (" C = ");
TEXT_IO.PUT_LINE (C);

TEXT_IO.PUT_LINE (" A = ");
J := 1;
for I1 in 1 .. 4 loop
  for I2 in 1 .. 3 loop
    INT_IO.PUT (A (I1, I2));
  end loop;
  TEXT_IO.NEW_LINE;
end loop;

end CALLFOR;
```

Example 9-5. UCS Ada Main Program to Call UCS FORTRAN Subprogram

```
subroutine forsub(i,c,a)
integer i
character*12 c
integer a(3,4)

print *, ' '
print *, '* In FORTRAN subprogram * '
print *, ' I = ', i
print *, ' C = ', c
print *, ' A = '

DO 10 j1 = 1,4
10  print *, (a(j2,j1), j2 = 1,3)

i = 456
c = '* FORTRAN *'

DO 20 j1 = 1,4
  DO 20 j2 = 1,3
20    a(j2,j1) = a(j2,j1) + j2 + (j1-1)*3

return
end
```

Example 9-6. UCS FORTRAN Subprogram Called from UCS Ada Main Program

UCS Ada Calling UCS FORTRAN

* In Ada main program *

I = 123

C = * Ada *

A =

1	2	3
4	5	6
7	8	9
10	11	12

* In FORTRAN subprogram *

I = 123

C = * Ada *

A =

1	2	3
4	5	6
7	8	9
10	11	12

* Back in Ada main program *

I = 456

C = * FORTRAN *

A =

2	4	6
8	10	12
14	16	18
20	22	24

Figure 9-3. Output of UCS Ada Program Calling UCS FORTRAN Subprogram

9.6. UCS Ada Calling UCS Pascal

Table 9-5 lists the data types defined by Ada and indicates the corresponding Pascal data type, if any.

Table 9-5. Ada vs Pascal Data Types and Attributes

Ada Actual Argument	Pascal Argument	Relation
integer (IN)	integer	YL
integer (OUT,IN OUT)	integer (VAR)	YL
float (IN)	real	WD
float (OUT,IN OUT)	real (VAR)	WD
long_float (IN)	(none)	2WD, YT
long_float (OUT,IN OUT)	(none) (VAR)	2WD, YT
access (IN)	pointer	YP
access (OUT,IN OUT)	pointer (VAR)	YP
enumeration (IN)	enumerated	YL, YV
enumeration (OUT,IN OUT)	enumerated (VAR)	YL, YV
boolean (IN)	boolean	YL
boolean (OUT,IN OUT)	boolean (VAR)	YL
character (IN)	integer	YL
character (OUT,IN OUT)	integer (VAR)	YL
string	string	Y
array	array	YL
record	record	YL

Legend

Y Argument type allowed with no restrictions.

N Argument type not allowed.

WD Size is word.

2WD Size is double word.

YL Argument type allowed, but data item must have proper storage layout.

YP Pointer can be passed, but Ada access type needs pragma specification.

YT Argument can be passed, but type may not exist in one of the languages and must be handled within called program.

YV Argument type allowed, but value passed must be within range required by called program.

You should be aware of the following considerations when calling a Pascal program from an Ada program:

- Ada passes arguments to Pascal by value (PRV), which is the default, or by reference (PBR). To pass by reference, the Pascal formal parameter declaration must include the keyword VAR.
Any IN scalar parameter is treated as PBV, and any OUT or IN OUT scalar parameter is treated as PBR. Therefore, always match a Pascal variable (VAR) parameter with an Ada OUT or IN OUT parameter.
- If data is modified in the Pascal subprogram, the Ada program should pass either of the following:
 - Access value pointing to the data
 - PBR parameter (OUT or IN OUT)
- Since the Pascal subprogram is being called from a program written in another language, the Pascal subprogram cannot use external files (including the standard files INPUT and OUTPUT), because external files must be declared in the main program and cannot be passed as arguments between these two languages. The example at the end of this subsection demonstrates this consideration. The Pascal subprogram does no input or output.
- If an Ada character string is passed to a Pascal variable-length string, the Ada program is responsible for simulating the internal representation of the Pascal variable-length string, which consists of the following parts:
 - First word, which contains the size in bits of the string
 - Rest of the word-aligned string data, which is allocated to the maximum sizeRefer to the UCS Pascal Programming Reference Manual for details.
- The Pascal type char is implemented as a 9-bit integral value, left-justified within a word. The Ada default for type character is implemented as a word integral (right-justified) value. Therefore, do not pass an Ada item of type character to a Pascal formal parameter of type char.

Example

This example demonstrates a call from an Ada main program to a Pascal subprogram, with the following actions by the main program and subprogram:

- The Ada main program initializes an integer scalar, a string, and a two-dimensional integer array. Each element of the array is initialized with its ordinal value in column-major order.
- The contents of the Ada data items are displayed to verify initialization.
- The Ada main program calls the Pascal subprogram, passing the three data items as actual arguments.
- The subprogram updates the contents of the dummy arguments and returns control to the main program. The changes to the string and scalar integer arguments are straightforward. The array is updated by adding the value of the

ordinal position of each element to the current contents of each element. Therefore, when the contents of the array are again displayed in the Ada main program, the value of each element should be double the original value.

- The main program again displays the contents of the arguments to verify that the Pascal subprogram updated them correctly.

Figure 9-4 shows the output from executing this program.

UCS Ada Calling UCS Pascal

```
-- This Ada main program calls a Pascal subprogram.

with TEXT_IO;

procedure CALLPAS is

    I, J : INTEGER;

    subtype STR_TYPE is STRING (1 .. 12);
    C : STR_TYPE;

    type ARR_TYPE is array (1 .. 4, 1 .. 3) of INTEGER;
    A : ARR_TYPE;

    procedure PASSUB (I1 : in out INTEGER;
                      C1 : in out STR_TYPE;
                      A1 : in out ARR_TYPE);
    pragma INTERFACE (UPAS, PASSUB);

    package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

begin
    TEXT_IO.PUT_LINE ("** In Ada main program **");

    -- Initialize and print variables

    I := 123;
    TEXT_IO.PUT (" I = ");
    INT_IO.PUT (I);
    TEXT_IO.NEW_LINE;

    C := "** Ada *";
    TEXT_IO.PUT (" C = ");
    TEXT_IO.PUT_LINE (C);

    TEXT_IO.PUT_LINE (" A = ");
    J := 1;
    for I1 in 1 .. 4 loop
        for I2 in 1 .. 3 loop
            A (I1, I2) := J;
            INT_IO.PUT (A (I1, I2));
            J := J + 1;
        end loop;
        TEXT_IO.NEW_LINE;
    end loop;

    -- Call UPAS
```

```

PASSUB (I, C, A);

-- Print variables on return from UPAS

TEXT_IO.NEW_LINE;
TEXT_IO.PUT_LINE ("** Back in Ada main program **");

TEXT_IO.PUT (" I = ");
INT_IO.PUT (I);
TEXT_IO.NEW_LINE;

TEXT_IO.PUT (" C = ");
TEXT_IO.PUT_LINE (C);

TEXT_IO.PUT_LINE (" A = ");
J := 1;
for I1 in 1 .. 4 loop
  for I2 in 1 .. 3 loop
    INT_IO.PUT (A (I1, I2));
  end loop;
  TEXT_IO.NEW_LINE;
end loop;

end CALLPAS;

```

Example 9-7. UCS Ada Main Program to Call UCS Pascal Subprogram

UCS Ada Calling UCS Pascal

```
COMPILATION UNIT uada_interlanguage_call_demo ;

TYPE
  real_array      = ARRAY[1..4,1..3] OF INTEGER ;
  char_array_type = PACKED ARRAY[1..12] OF CHAR ;

PROCEDURE ENTRY passub(VAR iparm : INTEGER;
                       VAR cparm : char_array_type ;
                       VAR aparm : real_array) ;

VAR
  i,j : INTEGER ;

BEGIN
  iparm := 456 ;
  cparm := '* Pascal * ' ;
  FOR i := 1 TO 4 DO
    FOR j := 1 TO 3 DO
      aparm[i,j] := aparm[i,j] + j + (i-1)*3 ;
    END.
  END.
```

Example 9-8. UCS Pascal Subprogram Called from UCS Ada Main Program

* In Ada main program *

I = 123

C = * Ada *

A =

1	2	3
4	5	6
7	8	9
10	11	12

* Back in Ada main program *

I = 456

C = * Pascal *

A =

2	4	6
8	10	12
14	16	18
20	22	24

Figure 9-4. Output of UCS Ada Program Calling UCS Pascal Subprogram